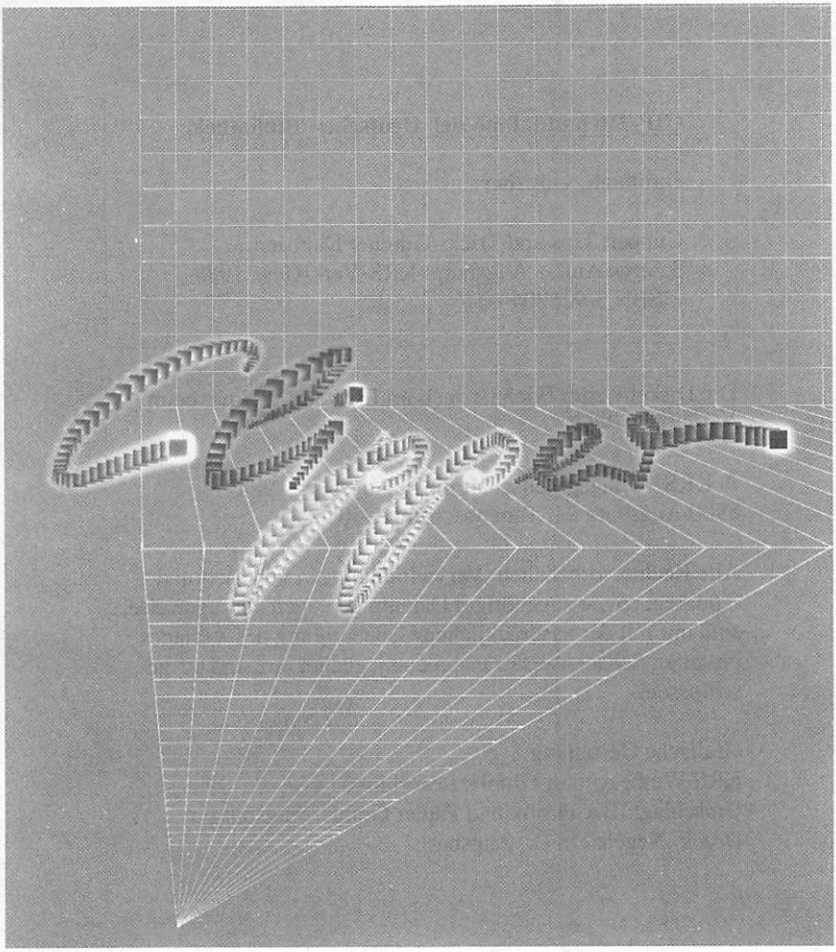
Clipper™ Tips und Tricks



Günther
Daubach

CIP-Titelaufnahme der Deutschen Bibliothek

Daubach, Günther:

Clipper, Tips und Tricks/Günther Daubach. -
3., erw. Aufl. - Augsburg: KRS-Verl.-Ges., 1988
ISBN 3-926204-90-7

Die Diskette zum Buch ist auch im IBM 3,5"-Format beim Verlag erhältlich.

© KRS Verlagsgesellschaft mbH, Augsburg 1988
8900 Augsburg 21, Tattenbachstraße 30

Alle Rechte vorbehalten. Vervielfältigungen von Text und Grafiken, sowie zugehöriger Datenträger, auch auszugsweise, sind nur mit der ausdrücklichen Genehmigung des Verlages gestattet. Schadensersatzansprüche jeglicher Art sind ausgeschlossen.

Grafische Gestaltung:
KRS Werbeagentur GmbH, Leverkusen
Umschlag: IBA Plastik und Papier GmbH, Augsburg
Druck: Negele Druck, Augsburg

Vorwort zur dritten Auflage

Dieses Buch richtet sich an zwei Zielgruppen, den "noch nicht"-Clipper Programmierer, der jedoch bereits Erfahrungen mit den Datanbanksystemen dBASE II oder dBASE III von Ashton Tate gesammelt hat und an den "schon"-Clipper Programmierer.

Für die erste Gruppe soll das Buch die wichtigsten Unterschiede zwischen dem Arbeiten mit Clipper und einem dBase-Interpreter aufzeigen und speziell die neuen Möglichkeiten demonstrieren, die man beim Einsatz des Clipper Compilers hat, ohne daß eine vollig neue Programmiersprache und Programmierart erlernt zu werden braucht.

Der zweiten Gruppe soll dieses Buch eine Sammlung von Ideen und Konzepten bieten, die bei der täglichen Arbeit mit Clipper und bei der Entwicklung eigener Anwendungen hilft. Hier dürften sicher besonders die "Hintergrund-Informationen", z.B. über den "Einbau" anwender-definierter Funktionen in Assembler- oder C-Sprache und die im gesamten Buch enthaltenen Programmiertricks wertvoll sein.

Das Buch ist aus der praktischen Arbeit mit Clipper entstanden und erhebt für sich nicht den Anspruch, den "Stein der Weisen" zu enthalten oder gar eine vollständige Sammlung aller Tricks und Möglichkeiten zu sein. Viele Programmbeispiele sind nicht bis zum letzten Befehl "ausgereizt" und lassen Spielraum für eigene Ideen und Änderungen.

Bei den Programmbeispielen wurde Wert auf eine möglichst übersichtliche Gliederung und "Anwendersicherheit" gelegt. Die Beispiele könnten zum Teil kompakter programmiert und auf Geschwindigkeit "getrimmt" werden, wobei aber dann die Übersichtlichkeit verlorengeht.

Auf der beiliegenden Diskette (IBM-Format, 360 KByte) sind alle Programmbeispiele als Quelldateien enthalten. Alle Beispiele sind außerdem in einer Bibliothek-Datei zusammengefaßt, so daß Leser ohne Assembler oder C-Compiler die Routinen ebenfalls ausprobieren und benutzen können. Alle Programme und die Bibliothek-Datei sind auf die Clipper-Version Sommer ´87 abgestimmt. Die in früheren Auflagen enthaltenen Varianten für ältere Clipper-Versionen wurden nicht mehr aufgenommen, da jedem Besitzer einer autorisierten Clipper-Version von Nantucket ein kostenloses Update angeboten wird. Die neue Clipper-Version ist gegenüber den älteren Fassungen deutlich verbessert worden, so daß ein Update unbedingt zu empfehlen ist.

In dieser überarbeiteten und erweiterten Auflage wurden auch alle Texte und Programme auf den Stand der Clipper-Version Sommer ´87 gebracht, so daß problemlos mit der nun aktuellen Fassung des Compilers gearbeitet werden kann.

Auf der Diskette ist auch der KRS Profi-Menü-Manager enthalten, den Sie als erstes mit dem Kommando `START <Return>` aufrufen sollten. Das Programm zeigt wichtige Zusatz-Informationen zum Buch und zur Diskette auf dem Bildschirm an.

An dieser Stelle möchte ich es nicht versäumen, all den Lesern der vorigen Auflagen zu danken, die mir wertvolle Hinweise für Verbesserungen und Erweiterungen gegeben haben und die mir natürlich auch geholfen haben, die unvermeidlichen Fehler zu beseitigen.

Einen besonderen Dank möchte ich aber auch an das Entwicklerteam von Nantucket richten dafür, daß sie den Clipper-Compiler geschaffen haben und dafür, daß sie sich auf ihren Lorbeeren nicht ausruhen, sondern mit jeder neuen Version (und wenn sie noch so lange auf sich warten läßt) immer wieder Verbesserungen einführen und für neue Überraschungen sorgen.

In der Hoffnung, daß dieses Buch allen dBASE-Programmierern zeigt, wie sie mit Clipper noch leistungsfähigere Programme entwickeln können, und allen Clipper-Programmierern hilft, noch mehr aus Clipper "herauszuholen" wünsche ich "glitchfree programming"!

Burscheid im Juni 1988 Günther Daubach

Inhalt

1	**Die dBASE-Datenbanksprache und der dBASE-Interpreter**	9
1.1	Was ist ein Interpreter?	11
1.2	Was ist ein Compiler?	14
1.3	Der Clipper Compiler	16
2	**Wie ein Clipper-Programm aufgebaut und compiliert wird**	21
2.1	Die Gliederung eines Clipper-Programms	21
2.2	Die Gliederung eines Clipper-Programms in Prozeduren	25
2.3	Wie ein Clipper-Programm eingegeben wird	26
2.4	Wie ein Clipper-Programm compiliert wird	28
2.5	Compilieren "auf Raten"	29
2.6	Das Hilfsprogramm MAKE von Microsoft	29
2.7	Das Linken eines Clipper-Programms	32
3	**Qverlays**	35
3.1	Der Overlay-Linker PLINK86	37
3.2	Beispielprogramm für eine Overlay-Struktur	38
	1. Bilden einer einzelnen EXE-Datei	40
	2. Bilden getrennter Haupt- und Overlay-Dateien	41
	3. Bilden einer Haupt-Datei und von Overlay-Dateien, die mehrere OBJ-Module enthalten	42
3.3	Strukturen mit mehreren Overlay-Bereichen	43
3.4	Zusammenfassung der PLINK86-Anweisungen zur Overlay-Struktur	46
3.5	Planung eines Clipper-Programms für die Verwendung mit Overlay-Strukturen	48
	1. Hauptmodul	48
	2. Untermodule	49
	3. Hilfsmodule	49
4	**Die Größe von Clipper-compilierten Programmen**	51
5	**Prozeduren**	55
5.1	Werte-Übergabe bei Prozeduren, Gültigkeit von Variablen	57
6	**Anwender-definierte Funktionen (UDFs - user defined functions)**	63
6.1	Was ist eine Funktion?	63
6.2	Wie wird eine Funktion definiert?	65
6.3	Private Variablen in Funktionen	66
7	**Anwender-definierte Funktionen in Clipper-Sprache**	71
7.1	Position einer Zeichenkette in einer anderen Zeichenkette	

7.2	Tastendruck lesen und solange den Cursor ausschalten	72
7.3	Anzeige einer Fehlermeldung mit Quittung	73
7.4	Text-Ausgabe mit Ja/Nein-Abfrage	74
7.5	Ausgabe eines Abfragetextes mit mehreren Antwortmöglichkeiten	76
7.6	Datum auf Gültigkeit prüfen	78
7.7	Zeit auf Gültigkeit prüfen	79
7.8	Prüfen auf "leere" String-Variable	81
7.9	Bereichsprüfung eines numerischen Wertes	82
7.10	Zeitdifferenz berechnen	83
7.11	Potenzberechnung	84
7.12	Zentrierte Ausgabe eines Textes	84
7.13	Rekursive Funktionen	85
8	**Anwender-definierte Funktionen in anderen Sprachen**	87
8.1	Aufrufe zur Parameter-Übergabe	87
8.2	Aufrufe zur Parameter-Rückgabe	88
8.3	Aufrufe zur Parameter-Prüfung	89
8.3.1	Funktion __PARINFO()	89
8.3.2	Funktion __PARINFA()	90
8.3.3	Funktion __PARCLEN()	90
8.3.4	Funktion __PARCSIZ()	91
8.4	Funktionen zur Speicherverwaltung	91
8.4.1	Anforderung von Speicher: __EXMGRAB()	91
8.4.2	Freigabe von Speicher: __EXMBACK()	91
9	**Anwender-definierte Funktionen in Assemblersprache**	93
9.1	Allgemeiner Aufbau einer UDF in Assemblersprache	93
9.2	Verwendung der Funktion __PARINFO()	95
9.2.1	Bestimmung der Parameter-Anzahl	95
9.2.2	Bestimmung der Parameter-Typen	96
9.3	Verwendung der Funktion __PARINFA()	96
9.3.1	Bestimmung der Elementzahl eines Arrays	96
9.3.2	Bestimmung des Typs eines Array-Elementes	97
9.4	Übernahme von Parametern	97
9.5	Aufruf von Assembler-UDFs aus einem Clipper-Programm	99
9.6	Universelle Makro-Datei für Assembler-UDFs	100
9.7	Sammlung nützlicher UDFs in Assemblersprache	108
9.7.1	Abfrage, ob der Cursor eingeschaltet ist	108
9.7.2	Töne bestimmer Höhe und Dauer erzeugen	109
9.7.3	Form des Cursors verändern	112
9.7.4	Bestimmen des aktiven Laufwerks	113
9.7.5	Aktuellen Pfad abfragen	114
9.7.6	Laufwerk prüfen	116
9.7.7	Bildschirm mit Zeichen und Attribut füllen	118

9.8	Grafik-Funktionen	123
9.8.1	Bildschirm-Modus umschalten	123
9.8.2	Grafikschirm löschen	125
9.8.3	Hintergrundfarbe setzen	126
9.8.4	Farb-Palette auswählen	127
9.8.5	Punkt setzen/löschen	129
9.8.6	Linie zeichnen	130
9.8.7	Beispiele für den Aufruf der Grafik-Funktionen	135
9.8.8	Grafik-Schirm in eine Datei speichern	137
9.8.9	Datei in den Grafikschirm lesen	139
9.8.10	Abschließende Bemerkungen zu den Grafikroutinen	141
10	**Anwender-definierte Funktionen (UFDs) in "C"**	143
10.1	Welcher C-Compiler "passt" zu Clipper?	143
10.2	Welche Compiler-Optionen müssen gesetzt werden?	143
10.3	Funktionen aus der Microsoft-C Library	144
10.4	Standard Header-Datei EXTEND.H	144
10.5	Parameter-Übergabe und Wert-Rückgabe Beispiel: Zeichenkettenvergleich mit "Joker-Zeichen" (Wildcards)	145
10.6	Suchen, ob eine bestimmte Gruppe von Zeichen an beliebiger Stelle einer Zeichenkette vorkommt	148
10.7	Zählen, wieviele Zeichen einer Gruppe von Zeichen an beliebiger Stelle in einer Zeichenkette vorkommen	150
10.8	Abfragen von Clipper-SET-Parametern	152
11	**Ein einfaches Testprogramm für Funktionen**	157
12	**Prüfen und Anlegen von Dateien**	159
12.1	Bestimmen des Schlüssel-Ausdruckes einer Index-Datei	163
12.2	Hilfsprogramm zum Generieren der Befehlsfolge zum Anlegen einer Dateistruktur	163
13	**Die kontext-abhängige Hilfe - HELP.PRG**	167
13.1	Wichtige Punkte, die bei HELP.PRG beachtet werden sollten	170
13.1.1	Bildschirm löschen	170
13.1.2	GET...READ innerhalb von HELP.PRG	170
13.1.3	Dateien innerhalb von HELP.PRG	170
13.2	Universelles HELP-Modul	171
13.3	Hilfeaufruf bei INKEY()	174
14	**Die SET KEY...TO... Anweisung**	177
15	**Die VALID-Klausel bei der GET-Anweisung**	185

16	**Anwendungen mit PROMPT...MENU TO**	183
16.1	"Pull down"-Menüs	189
16.2	MENU TO "zweckentfremdet" zur Satzauswahl	194
17	**Die Clipper-Bibliothekdatei CLIPPER.LIB**	199
17.1	Positionieren des Cursors _SETCURS	199
175.2	Cursor ein- und ausschalten	200
17.3	Rollen und Löschen eines Bildschirm-Fensters	200
17.4	Aktivieren eines Ausgabefensters	201
17.5	Direkte Drucker-Ausgabe	202
17.6	Interne reservierte Wörter	202
18	**Verwalten selbst definierter Routinen**	213
18.1	Das Anlegen und Verwalten eigener Bibliothek-Dateien (Libraries)	214
19	**Was sonst nirgendwo hineinpasst...**	217
19.1	"Mehrdimensionale" Arrays	217
19.2	Papier-Sparroutine	218
19.3	Programm-Überwachung und Sicherheit	220
19.4	"Count-Down" Anzeige bei Transaktionen	225
19.5	"Anti-PACK" Routinen	227
19.6	"Verzeigerte" Dateien	229
19.7	Die ungleichen Zwillinge BOF() und EOF()	241
19.8	Bilden von Index-Ausdrücken	242
19.9	Eine einfache Taschenrechner-"Po-Up"-Routine	244
20	**Programm-Beispiele**	253
20.1	Clipper-BROWSE	253
20.2	Clipper-CREATE	259
21	**Rückgabecodes von INKEY() und LASTKEY()**	269
22	**Warenzeichen**	273
23	**Stichworte**	275

1 Die dBASE-Datenbanksprache und der dBASE-Interpreter

Nachdem Microcomputersysteme mit relativ preiswerten Massenspeichern, wie Diskettenlaufwerken und später Festplattenspeichern ausgestattet wurden und außerdem zu relativ günstigen Preisen ein ausreichend großer Arbeitsspeicher realisiert werden konnte, haben verschiedene Software-Entwickler Programme geschaffen, um Massendaten mit solchen Systemen zu speichern und zu verarbeiten.

Es lag natürlich nahe, die für Großrechner entworfenen Programmiersprachen, wie *COBOL* oder *RPG* auf Microcomputersysteme zu übertragen, da diese Sprachen speziell für die Verarbeitung von Massendaten geschaffen wurden. Heute sind einige recht leistungsfähige Versionen verfügbar, die jedoch keine allzugroße Verbreitung gefunden haben. Der wichtigste Grund dafür ist sicherlich, daß ein Großteil der Microcomputer nicht von professionellen Programmierern, sondern von "Durchschnitts-Anwendern" benutzt wird.

Die wichtigste Forderung, die ein solcher Anwender an seinen persönlichen Computer - den *Personalcomputer* - stellt, ist, daß er ohne großen Zeitaufwand und ohne umfangreiche Vorkenntnisse in der Lage sein will, Daten zu erfassen, zu speichern, nach bestimmten Kriterien zu ordnen und zu finden, sowie auszuwerten und auszudrucken.

Ein Programm, das dem Anwender diese Anforderungen erfüllt, muß folgende wesentliche Kriterien erfüllen:

- Einfaches Anlegen von Dateistrukturen mit der Möglichkeit der nachträglichen Änderung ohne wesentlichen Verlust bereits erfaßter Daten.
- Einfache Möglichkeit, Daten interaktiv über ein Bildschirm-Terminal einzugeben und zu ändern.
- Möglichst schnelles Auffinden gespeicherter Daten durch geeignete Struktur der Datenbank mit entsprechenden Suchschlüssen.
- Möglichkeiten zur übersichtlichen Anzeige der erfaßten Daten auf dem Bildschirm-Terminal.
- Einfacher Ausdruck der erfaßten Daten in Tabellenform in verschiedenen Formaten und in verschiedenen Gruppierungen.

- Möglichkeit zur Selektion bestimmter Daten aus dem Gesamtbestand nach unterschiedlichen Kriterien.

- Auswertung der Daten nach verschiedenen Gesichtspunkten, z.B. Addition bestimmer Feld-Inhalte oder Zählen bestimmter Daten.

- Möglichkeit, bestimmte, immer wiederkehrende Abläufe einmal zu erfassen und danach automatisch auszuführen.

Die letzte Eigenschaft der obigen Aufstellung bedeutet nichts anderes, als daß es möglich sein muß, das Datenbanksystem zu programmieren.

Es hat verschiedene Programm-Entwicklungen gegeben, die diese Forderungen mehr oder weniger gut erfüllt haben. Das sicherlich weltweit am meisten verbreitete Datenbankprogramm ist dBASE, das von der Firma Ashton Tate vertrieben wird.

Das Programm wurde zuerst in der Version dBASE II für 8-Bit Microcomputer unter dem Betriebssystem CP/M in den Markt gebracht. Es ist kaum zu schätzen, wieviele Kopien dieses Programms, die leider nicht immer legal angefertigt wurden, auf der ganzen Welt existieren.

Mit der Verfügbarkeit von 16-Bit Computern wurde dBASE II auch auf die Betriebssysteme CP/M 86 und MS-DOS bzw. PC-DOS übertragen. Wie bei vielen Programmen, die "schnell" von 8- auf 16-Bit-Systeme übertragen wurden, war diese Version von dBASE II nicht in der Lage, den verfügbaren Arbeitsspeicher und die Eigenschaften der neuen Mikroprozessoren (8088, 8086 usw.) auszunutzen.

Eine wesentliche Verbesserung brachte die Version dBASE III, die für die 16-Bit Systeme "maßgeschneidert" wurde. Zum Glück hat jedoch Ashton Tate dafür gesorgt, daß die neue Fassung, soweit möglich, "aufwärtskompatibel" zu dBASE II ist. Das bedeutet, daß die unzähligen existierenden Anwendungen unter dBASE II ohne allzugroßen Aufwand auf dBASE III übertragen werden konnten. Natürlich werden dabei keine neuen Eigenschaften von dBASE III genutzt, aber alleine die höhere Verarbeitungsgeschwindigkeit und die erhöhte Sicherheit des Programms rechtfertigen diesen Schritt.

dBASE erlaubt es dem Anwender die "eingebauten" Eigenschaften über relativ einfach zu erlernende englische Kommandos aufzurufen und auszuführen. Mit diesem "Vorrat" an Kommandos werden alle in der obigen Liste aufgeführten

wichtigen Kriterien für ein Datenbankprogramm erfüllt - dBASE geht in einigen Punkten sogar darüber hinaus.

Damit alle Kommandos interaktiv ausgeführt werden können, muß dBASE als Interpreter aufgebaut sein. Bei einem Interpreter wird eine über die Tastatur eingegebene Kommandozeile unmittelbar nach erfolgter Eingabe analysiert, auf Richtigkeit geprüft, in interne Maschinenbefehle umgesetzt und ausgeführt. Dadurch ist es möglich, daß der Anwender nach Eingabe eines Kommandos sofort das gewünschte Ergebnis erhält. Es kann natürlich auch ein unerwünschtes Ergebnis oder sogar eine Fehlermeldung erscheinen, wenn ein unzutreffendes oder fehlerhaftes Kommando eingegeben wurde. In diesem Fall kann der Anwender seine Eingabe sofort korrigieren und einen erneuten Versuch machen.

Soll das System programmiert werden, dann erstellt der Anwender eine Textdatei, in der die Kommandos in der gleichen Reihenfolge enthalten sind, wie er sie in der interaktiven Betriebsart über die Tastatur eingeben würde. Natürlich sind weitere Kommandos erforderlich, damit sinnvoll ein Programm erstellt werden kann. Dazu gehören besonders Anweisungen zur Ablaufsteuerung, wie z.B. Anweisungen für Programmschleifen, Verzweigungen und logische Entscheidungen. dBASE erlaubt es, ein Programm in mehrere Textdateien aufzuteilen. Damit läßt sich ein Programm in kleinere übersichtliche Abschnitte untergliedern und öfters benötigte Programm-Abschnitte brauchen nur einmal geschrieben zu werden.

1.1 Was ist ein Interpreter?

Wenn dBASE ein Programm ausführt, dann werden die Textdateien mit den Programm-Anweisungen Zeile für Zeile gelesen, analysiert, auf Richtigkeit überprüft, in den internen Maschinencode übersetzt und ausgeführt. Der Vorgang ist ähnlich dem interaktiven Modus, nur daß hier die Anweisungen nicht von der Tastatur, sondern aus Dateien gelesen werden.

Der große Vorteil dieser Arbeitsweise ist sicherlich der, daß Korrekturen im Programm (z.B. beim Auftreten eines Fehlers) schnell und einfach durchgeführt werden können und danach das Programm sofort erneut ausgeführt werden kann.

Da ein Interpreter bei jedem Programmlauf den oben beschriebenen Vorgang (Analyse, Prüfung, Übersetzung und Ausführung) erneut erledigen muß, liegt es auf der Hand, daß diese Arbeitsweise Einbußen bei der Verarbeitungsge-

schwindigkeit bringen muß. Besonders in Programmschleifen, die sehr oft durchlaufen werden, ist es nachteilig, daß die darin enthaltenen Anweisungen immer wieder neu der Prozedur von der Analyse bis zur Ausführung unterzogen werden.

Die in dBASE enthaltene Datenbank-Programmiersprache besitzt einen großen Leistungsumfang und ermöglicht es, mit relativ kurzem Zeitaufwand leistungsfähige Anwendungen zu entwickeln. Es ist daher nicht verwunderlich, daß dBASE von einer großen Zahl von Programmierern als Werkzeug zur Erstellung "maßgeschneiderter" Programme für die unterschiedlichsten Einsatzbereiche verwendet wird. Damit ist dBASE von einem "Jedermann-Datenbanksystem" in den Bereich der Anwendungs-Programmiersprachen aufgestiegen.

Programmierer, die mit den entwickelten Programmen ihr Einkommen bestreiten wollen, stellen natürlich weitergehende Anforderungen an ein Programmierwerkzeug:

- Das an den Kunden ausgelieferte Programm soll sicher gegen Fehlbedienungen sein und es muß verhindert werden, daß das Programm oder der Datenbestand durch unerfahrende Benutzer verändert werden kann.

- Die geistige Leistung des Programmierers soll sein Eigentum bleiben, d.h., der Programmcode sollte von Dritten nicht eingesehen werden können.

- Das Programm sollte ein möglichst gutes Zeitverhalten haben.

- Das Programm sollte möglichst preisgünstig sein.

Die oben aufgezählten Forderungen sind durch ein interaktives System mit Interpreter nur schwer zu erfüllen.

Bei einem Interpreter ist der Anwender in der Lage, mit bestimmten Kommandos das Programm abzubrechen, direkt den Inhalt von Dateien zu verändern und Änderungen an Programmen vorzunehmen. Dies kann bei unsachgemäßer Anwendung zu gravierenden Verlusten von Daten und zu Fehlfunktionen der Programme führen. Nicht wenige Programmierer kennen sicher die Situation, daß sich ein Kunde über das fehlerhafte Arbeiten eines Programms beschwert, wobei sich dann herausstellt, daß der Kunde durch Unkenntnis Daten gelöscht oder geändert oder sogar das Programm "umgebaut" hat. Die Beweislast ist in diesen Fällen sehr schwer.

Da der dBASE-Interpreter Programme nur als reine Textdateien - gewissermaßen im "Klartext" - bearbeiten kann, besteht keine Möglichkeit, den Programmcode gegen Einsichtnahme durch Dritte zu schützen, der Programmierer muß zwangsläufig seinen Programmcode offen an den Anwender übergeben.

Das Zeitverhalten eines interpretierten Programmes wurde bereits weiter oben besprochen. Besonders bei großen Anwendungen kann ein interpretiertes Programm nicht gegenüber Programmen bestehen, die in anderen Sprachen entwickelt wurden.

Bei interpretierten Programmen ist zur Ausführung stets der Interpreter - in diesem Fall das Programm dBASE II oder dBASE III - erforderlich. Da Ashton Tate die Programme als Einzel-Lizenzen vertreibt, ist es nötig, daß jeder Anwender einer Applikation eine dBASE Programmlizenz rechtmäßig erwirbt. Der Programmierer ist nicht berechtigt, eine Kopie von dBASE an seine Kunden weiterzugeben - er würde damit gegen das Urheberrecht verstoßen. Daraus ergeben sich für den Anwender zusätzliche Kosten, die von vielen nicht aktzeptiert werden.

Natürlich hat auch Ashton Tate diese Problematik erkannt. Aus diesem Grund werden sogenannte "Run Time"-Pakete für dBASE angeboten. Allerdings wird damit die geschilderte Problematik nur teilweise gelöst.

Bei einer Run Time-Version handelt es sich um einen modifizierten Interpreter, der keinen Kommando-Modus mehr ausführt, d.h., es werden keine Anweisungen von der Tastatur angenommen. Außerdem ist die Run Time-Version in der Lage, verschlüsselte Programmdateien zu verarbeiten. Dazu erhält der Programmierer ein Hilfsprogramm, mit dem er die "Klartext"-Programme in verschlüsselte Programmdateien umwandeln kann. Dieser Programmschutz ist jedoch nicht sehr zuverlässig. Der Autor selbst hat aus "sportlichem Ehrgeiz" einen dBASE II Run Time-Decodierer geschrieben.

In den verschlüsselten Programmdateien werden die Anweisungen durch Kürzel (sogenannte "Tokens") dargestellt. Dadurch ist der Zeitaufwand für die Analyse einer Programmzeile natürlich kürzer, so daß sich ein etwas besseres Zeitverhalten als beim Interpreter ergibt.

Da die Run Time-Versionen preisgünstiger abgegeben werden, als der Interpreter, wirkt sich das natürlich positiv auf die Kostenbetrachtung aus.

1.2 Was ist ein Compiler?

Ein Compiler ist ein Hilfsprogramm, mit dem ein im "Klartext" vorliegender Programmcode, das *Quellprogramm*, in eine Datei übersetzt wird, die direkte Maschinen-Anweisungen enthält. Diese Datei kann vom Betriebssystem aus geladen und als Programm unmittelbar ausgeführt werden.

In der Praxis erfolgt diese Umsetzung in mehreren Schritten. Zunächst liest der Compiler das Quellprogramm Zeile für Zeile und analysiert dabei jede Zeile auf Fehler. Gleichzeitig legt er fest, welcher Speicherplatz für die Speicherung von Daten im Arbeitsspeicher benötigt wird und erzeugt entsprechende Strukturen zur Steuerung des Programms. Da in einem Programm in der Regel bestimmte Kommandos und Funktionen mehrfach an den unterschiedlichsten Stellen aufgerufen werden, erzeugt der Compiler nicht sofort den ausführbaren Maschinencode. Stattdessen werden in eine Zwischendatei - die *Objektdatei* - Aufrufe entsprechender Programmroutinen geschrieben. Diese Objektdatei enthält weiterhin Informationen über die verwendeten Namen von Variablen, den dazu benötigten Speicherplatz usw.

Je nach Struktur des Programms können einzelne Quelldateien getrennt mit dem Compiler übersetzt werden, so daß abschließend mehrere Objektdateien vorhanden sind.

In einem zusätzlichen Schritt werden mit einem weiteren Hilfsprogramm - dem *Linker* - die Objektdateien gelesen und miteinander verbunden. Wesentliche Aufgabe des Linkers ist es jedoch, aus einer sogenannten "*Bibliothek-Datei*" oder "*Library*" den Maschinencode für die Routinen zu lesen, für die der Compiler in die Objektdatei Aufrufe eingefügt hat. Es ist auch möglich, daß Routinen aus mehreren Bibliothek-Dateien gelesen werden.

Nachdem der Linker alle erforderlichen Programmteile und Routinen "zusammengebaut" hat, erzeugt er eine Datei, die ein eigenständig vom Betriebssystem ausführbares Maschinenprogramm enthält.

Es liegt auf der Hand, daß ein mit dem Compiler übersetztes Programm die wichtigen Anforderungen eines Programmierers erfüllt:

- Das Programm verhindert bei entsprechender Programmierung, daß Fehlbedienungen durch den Anwender größere Schäden an Datenbeständen verursachen können.

- An den Anwender braucht nur das eigenständig lauffähige Maschinenprogramm, nicht aber der Quellcode ausgeliefert zu werden. Ein Rückschluß auf den Text des Quellprogramms ist aus dem Maschinenprogramm - falls überhaupt - nur mit einem erheblichen, wirtschaftlich nicht vertretbarem Aufwand möglich und das auch nur durch einen erfahrenen Programmierer, der in dieser Zeit sicherlich eine ähnliche Anwendung selbst programmieren könnte.

- Da die Umsetzung in den Maschinencode einmalig durch den Compiler vorgenommen wird und nicht erst bei der Programm-Ausführung erfolgt, ist das Zeitverhalten bei einem compilierten Programm deutlich günstiger als bei einem interpretierten Programm.

- In der Regel wird vom Lieferanten des Compilers nur eine einmalige Lizenzzahlung für den Compiler verlangt. Die mit dem Compiler erzeugten Anwendungen sind in unbegrenzter Stückzahl frei von zusätzlichen Gebühren, so daß Anwendungen kostengünstig an den Kunden abgegeben werden können.

Ein weiterer wichtiger Vorteil eines Compilers ist es, daß bei der Übersetzung eines Programms bereits Fehler erkannt werden, die bei einem Interpreter unter Umständen lange Zeit verborgen bleiben. Wird z.B. die Struktur einer Programmschleife falsch aufgebaut, so erkennt dies der Compiler bereits bei der Übersetzung. Beim Interpreter tritt der Fehler erst dann auf, wenn die Schleife tatsächlich ausgeführt werden soll. Falls diese nur in wenigen Sonderfällen erreicht wird, kann es lange dauern, bis man eine solche "Zeitbombe" entdeckt. Es ergibt sich daher besonders beim Interpreter die Notwendigkeit, ein Programm bis "in den letzten Winkel" zu testen, bevor es in den "scharfen" Einsatz geht.

Je länger ein Programmierer mit einer Programmiersprache arbeitet, desto mehr Programmroutinen wird er entwickeln, die er in den unterschiedlichsten Anwendungen einsetzen kann - vorausgesetzt, er beachtet gewisse Systematiken beim Entwurf solcher Routinen. Ein Compiler bietet den Vorteil, daß diese Routinen, einmal compiliert, als Objektdateien gespeichert und bei Bedarf mit anderen Programmteilen zusammengebunden werden können. Dadurch erübrigt sich ein erneutes Compilieren und der damit verbundene Zeitaufwand. Es ist sogar möglich, mit einem Hilfsprogramm - einem *Library Manager* - die vorhandenen Objektmodule in einer eigenen Bibliothekdatei zusammenzufassen, aus der der Linker die jeweils erforderlichen Routinen liest und in ein Programm einbindet.

Es soll an dieser Stelle ein Nachteil des Compilers nicht verschwiegen werden: jede Programmänderung erfordert einen erneuten Übersetzungs- und Linklauf. Die dafür benötigte Zeit kann - besonders bei großen Anwendungen - erheblich sein. Es ist daher unbedingt nötig, daß bei Einsatz eines Compilers besonderer Wert auf die Konzeption und Strukturierung eines Programms gelegt wird, bevor man auch nur eine Programmzeile des Quellcodes eingibt. Letztendlich ist diese Vorgehensweise jedoch sicher besser, als wenn man mit dem Interpreter "mal eben" ein Programm ohne große vorherige Planung eingibt und Schritt für Schritt versucht, es zum Laufen zu bringen.

1.3 Der Clipper Compiler

Im Gegensatz zu anderen Programmiersprachen, wie z.B. BASIC oder Pascal nimmt die dBASE-Sprache dem Programmierer bei der Entwicklung von Datenbankprogrammen besonders die "Knochenarbeit" ab, Dateistrukturen, schnelle Suchfunktionen, Routinen zur formatierten Bildschirm-Ein- und Ausgabe und zum formatierten Ausdruck zu schreiben. Im Gegensatz zu anderen Programmiersprachen, die besonders für diese Belange geschaffen wurden, wie z.B. COBOL, verfügt die dBASE-Sprache über Elemente, die das strukturierte Programmieren ermöglichen und teilweise sogar erzwingen. Hierzu gehört das Unterteilen eines Programms in kleine, überschaubare Prozeduren mit der Möglichkeit, Parameter zu übergeben und lokale Variablen zu definieren und die Steuerung des Programm-Ablaufes mit `WHILE`-Schleifen, `CASE`- und `IF...THEN...ELSE`-Anweisungen.

Es liegt natürlich nahe, für eine solche Sprache einen Compiler zu entwickeln, um damit dem Programmierer ein leistungsfähiges Werkzeug in die Hand zu geben. Das amerikanische Softwarehaus *Nantucket* hat im Jahr 1984 seinen ersten dBASE III-Compiler für IBM- und IBM-kompatible Personalcomputer vorgestellt und ihm den Namen *Clipper* gegeben. Bis heute ist dieser Compiler mehrfach verbessert und erweitert worden. In der aktuell vorliegenden Version (Sommer ´87) enthält er als Standard auch Funktionen und Kommandos zur Programmierung von Anwendungen, die in einer Netzwerk-Umgebung eingesetzt werden können.

Clipper ist ein "echter" Compiler, d.h., er übersetzt ein Quellprogramm unmittelbar in ein linkbares Objektprogramm, das mit einem Linker und der mitgelieferten Clipper-Bibliothekdatei zu einem lauffähigen Maschinenprogramm gebunden werden kann.

Andere Produkte wandeln das Quellprogramm in einen Zwischencode um, der dann von einem speziellen Run Time Modul interpretiert und ausgeführt wird. Dieser Zwischencode ist zwar wesentlich kompakter als der von dBASE Run Time verarbeitete Code und wird damit schneller ausgeführt, die Ausführungsgeschwindigkeiten eines mit Clipper compilierten Programms können jedoch damit nicht erreicht werden.

Das vordringlichste Ziel der Entwickler von Clipper war es, einen Compiler zu schaffen, der den Code existierender dBASE III Programme ohne Änderung "versteht", um so dem Programmierer die Möglichkeit zu geben, ein bereits geschriebenes Programm schnell umzuwandeln. Dieses Ziel kann nicht in allen Fällen erreicht werden. So verfügt Clipper nicht über die interaktiven Kommandos (es handelt sich ja eben *nicht* um einen Interpreter) und es stehen demnach auch nicht die dBASE III Kommandos zur Verfügung, die vorwiegend interaktiv verwendet werden, jedoch auch in einem Programm enthalten sein können. Dazu gehören besonders:

- EDIT - Editieren eines vorhandenen Datensatzes in einer Standard- oder in einer durch eine Format-Datei definierten Maske.
- APPEND - Anfügen eines neuen Datensatzes in einer Standard- oder in einer durch eine Format-Datei definierten Maske.
- BROWSE - Tabellarische Anzeige mehrerer Datensätze mit der Möglichkeit, durch den Datenbestand zu "blättern" und ggf. Änderungen, Löschungen und Neueingaben vorzunehmen.

Da diese Kommandos relativ feste Bildschirm-Anzeigen vorgeben, wurden sie von den meisten Programmierern auch in dBASE Programmen nicht verwendet, so daß ihr Fehlen im Compiler nicht allzu schwerwiegend ist. Darüberhinaus können sie durch "Ersatzroutinen" in Clipper relativ leicht nachgebildet werden. Ab der Version Herbst ´86 des Clipper Compilers ist eine Funktion DBEDIT() enthalten, mit der man ähnliche Möglichkeiten, wie bei BROWSE hat. Zusätzlich besteht bei DBEDIT() die Möglichkeit, die Reaktion auf bestimmte Tasteneingaben individuell zu programmieren. In der Version Sommer ´87 sind eine Reihe wertvoller Erweiterungen und Verbesserungen vorgenommen worden. Der neue Compiler basiert nun auf Microsoft C, wodurch eine deutliche Geschwindigkeitssteigerung erreicht wurde.

Wesentlich ist neben der Kompatibilität der Programmiersprache jedoch auch die Kompatibilität der verwendeten Dateien. Der Clipper Compiler verwendet bei allen Dateiarten - außer bei Index-Dateien - dieselbe Struktur wie dBASE III, so daß vorhandene Datenbestände, die unter dBASE III erfaßt wurden, ohne Probleme übernommen werden können. Die Entwickler von Clipper haben sich,

sicher nach reiflicher Überlegung, dazu entschlossen, die Struktur von Index-Dateien zu ändern, um eine schnellere Bearbeitung zu ermöglichen. Diese Inkompatibilität ist jedoch nicht allzu schwerwiegend, da die Index-Dateien automatisch neu aufgebaut werden können, ohne daß dazu langwierige manuelle Korrekturarbeiten erforderlich sind. Ab Version Sommer ´87 wird ein Zusatzmodul ausgeliefert, das die Verwendung dBase III-kompatibler Indexdateien gestattet. Allerdings werden diese Dateien langsamer verarbeitet und benötigen mehr Platz als die Clipper-Indexdateien.

Neben den Möglichkeiten, die die dBASE III Sprache bietet, besitzt Clipper demgegenüber eine Vielzahl von Erweiterungen. Hier kann man deutlich erkennen, daß die Entwickler von Clipper endlich die Funktionen, Kommandos und Eigenschaften realisieren konnten, die sie sich bei dBASE III schon lange gewünscht haben. Bisher hat jede neue Clipper-Version wieder einige Überraschungen in dieser Richtung geboten. Besonders auf diese Erweiterungen und die dadurch gebotenen Möglichkeiten soll in den folgenden Kapiteln des Buches eingegangen werden. Hier nur eine Aufstellung der wichtigsten Eigenschaften:

- Definition eigener Funktionen in Clipper-, C- oder Assemblersprache und damit Erweiterungsmöglichkeit der Programmiersprache nach eigenen Wünschen.
- Größere Anzahl von Feldern je Datensatz - maximal 1024.
- Erweiterung der `GET`-Anweisung durch die sehr leistungsfähige `VALID`-Klausel.
- Integrierter einfacher Text-Editor zur Bearbeitung von Memo-Feldern und Zeichenketten innerhalb eines Programmes (`MEMOEDIT()`). Ab Version Sommer ´87 wurde die Funktion mit wesentlichen Erweiterungen ausgestattet.
- Zeichenketten bis zu 64 KByte Länge.
- Interrupt-ähnlich gesteuerter Aufruf von Prozeduren über Funktionstasten und als Sonderfall hierbei die Möglichkeit, ein kontext-abhängiges Hilfesystem in die Applikation zu integrieren.
- Eindimensionale Arrays als Speichervariablen, die das tabellarische Bearbeiten von Werten vereinfachen.

Mittlerweile sind Anpassungen des Clipper-Compilers für MS-DOS Computer auf dem Markt, die nicht IBM-kompatibel sind. Da nur für wenige dieser Computer dBASE III verfügbar ist, bietet Clipper die einzigartige Möglichkeit, Programme, die ursprünglich für den IBM-PC entwickelt wurden, auf diese Systeme zu übertragen.

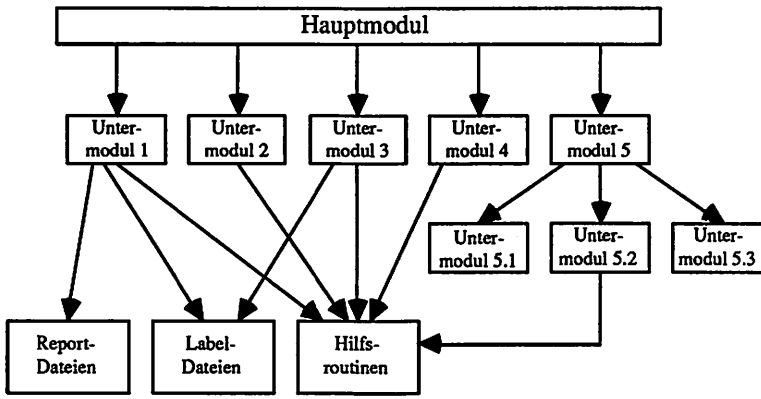
Durch die offene Struktur des Clipper-Compilers wurden externe Softwarehäuser zur Entwiclung von Erweiterungspaketen angeregt. Im deutschen Markt sind besonders Tom Rettig´s Library mit einer vielseitigen Sammlung nützlicher Routinen und die Programmierer-Tools von KRS bekannt. Die KRS-Tools umfassen die Super-Toolbox, eine Sammlung nützlicher Routinen mit besonderem Schwerpunkt auf Fenstertechnik, die Grafik-Toolbox mit Routinen für statistische und technische Grafiken und das Hilfe-Modul, mit dem in eine Applikation mühelos ein leistungsfähiges On-line Hilfesystem integriert werden kann.

Wenn man die erweiterten Möglichkeiten von Clipper in einem Programm einsetzt, so ist dieses Programm natürlich unter dBASE III nicht mehr lauffähig. Es besteht dann nicht mehr die Möglichkeit, die Programme zunächst mit dem Interpreter zu testen und abschließend zu übersetzen. In diesem Fall arbeitet man mit Clipper wie mit jeder anderen reinen Compilersprache auch. Wenn man sich aber erst einmal an diese Arbeitsweise gewöhnt hat, wird man wahrscheinlich dBASE III immer seltener verwenden.

2 Wie ein Clipper-Programm aufgebaut und compiliert wird

2. 1 Die Gliederung eines Clipper-Programms

Für fast alle Anwendungen läßt sich für ein Programmsystem eine gleichartige Grundstruktur festlegen:



Das Hauptmodul ist das Modul, mit dem das Gesamtprogramm gestartet wird und über das es - bei ordnungsgemäßer Programmierung auch wieder verlassen wird. Dieses Modul führt allgemeine Initialisierungen durch und es sollte auch eine Überprüfung vornehmen, ob alle Dateien der Datenbank vorhanden sind und gegebenenfalls fehlende Dateien anlegen. Schließlich enthält das Hauptmodul in der Regel einen Teil, über den der Anwender die verschiedenen Programmfunktionen auswählen kann. Diesen Programmteil bezeichnet man als "*Menü*" oder "*Funktionsauswahl*". Aus diesem Programmteil werden die Untermodule aufgerufen.

In der nächsten Ebene unter dem Hauptmodul findet man meistens mehrere Untermodule, in denen jeweils ein sinnvoll zusammengehöriger Teil des Programms enthalten ist. Je nach Komplexität des Programms kann es sinnvoll sein, die Programmfunktionen weiter zu untergliedern. Dadurch ist es möglich, daß aus den Untermodulen weitere Programmteile aufgerufen werden. Hierbei

können dann auch in den Untermodulen "*Submenüs*" enthalten sein, aus denen die Unter-Unterfunktionen der nächsttieferen Ebene aktiviert werden.

Im obigen Schema sind solche Unter-Untermodule dargestellt. Natürlich ist hier keine Grenze gesetzt. Es ist durchaus möglich, daß weitere untergeordnete Modul-Ebenen vorhanden sind.

Sehr oft werden bestimmte Programmteile, wie z.B. das Ausgeben einer Fehlermeldung, das Anzeigen einer Standard-Bildschirmmaske oder eine Routine zur Prüfung von Eingabewerten in vielen Programmteilen benötigt. Es liegt daher nahe, diese "*Hilfsroutinen*" in weiteren Programmteilen zusammenzufassen. Bei Clipper ist es besonders einfach möglich, daß diese Hilfsroutinen so vielseitig gestaltet werden, daß sie in den unterschiedlichsten Anwendungen benutzt werden können. Der Programmierer kann diese Routinen in einer eigenen Bibliothek-Datei zusammenfassen. Dadurch wird vermieden, daß diese Routinen bei jeder Programmentwicklung erneut compiliert werden müssen.

Wie bei dBASE können auch bei Clipper die Formate für die Ausgabe von Listen auf dem Bildschirm oder Drucker in *Reportdateien* (`.FRM`) definiert werden. Die Ausgabe einer solchen Liste wird im Programm mit der Anweisung `REPORT FORM <Report-Datei>` erreicht. Ebenso lassen sich die Formate von auszudruckenden Etiketten in *Labeldateien* (`.LBL`) definieren. Im Programm wird die Ausgabe von so definierten Etiketten mit der Anweisung `LABEL FORM <Label-Datei>` erreicht. In obigem Schema sind entsprechende Report- und Labeldateien enthalten.

Das Schema stellt die Gliederung eines Programms nach logischen Abschnitten dar. In dBASE wenden die meisten Programmierer dieses Gliederungsschema auch beim Aufbau der Programmdateien an. Jedes Modul - das Hauptmodul und alle Untermodule - werden in getrennten Programmdateien (`.PRG`) gespeichert. Für Hilfsroutinen werden gerne Prozedurdateien geschrieben, auf die dann aus anderen Programmteilen mit der Anweisung `SET PROCEDURE TO <Prozedurdatei>` zugegriffen werden kann.

Dieses bewährte Gliederungsschema sollte auch beim Arbeiten mit Clipper beibehalten werden, da so das Quellprogramm übersichtlich strukturiert werden kann. Ein weiterer Grund für die Aufteilung in Einzeldateien bei dBASE ist, daß damit die Bearbeitungsgeschwindigkeit um einen gewissenen Faktor erhöht werden kann. Bei Clipper wirkt sich allerdings die Aufteilung in mehrere Programmdateien in keiner Weise auf die spätere Verarbeitungsgeschwindigkeit aus.

Als Grundlage für die weiteren Betrachtungen soll das Programmdatei-Schema einer einfachen Adressverwaltung dienen:

Programm-Ebene	Dateiname	Erläuterung
Hauptmodul:	ADRVERW.PRG	
Untermodul 1:	NEUEIN.PRG	Eingabe neuer Adressen
Untermodul 2:	SUCHEN.PRG	Suchen von Adressen
Untermodul 3:	AENDERN.PRG	Ändern von Adressen
Untermodul 4:	LOESCHEN.PRG	Löschen von Adressen
Untermodul 5:	LISTEN.PRG	Adresslisten und -aufkleber
Untermodul 5.1:	ANZEIGE.PRG	Adressliste anzeigen
Untermodul 5.2:	DRUCK.PRG	Adressliste ausdrucken
Untermodul 5.3:	ETIKETT.PRG	Adressaufkleber drucken
Hilfsroutinen	HROUT.PRG	Prozedur-Datei mit den Prozeduren
	PLZTEST	Prüfung auf gültige Postleitzahl
	RAHMEN	Anzeige eines Standard-Bildschirms
	DRUCK_OK	Abfrage, ob Drucker bereit
	ADR_GET	Einlesen der Adress-Informationen für Neueingabe oder Änderung.

Das Hauptmodul ADRVERW könnte folgenden Aufbau haben:

```
*
* Allgemeine Definitionen
*
* < entsprechende Anweisungen >

SET PROCEDURE TO hrout     && in Clipper nicht unbedingt nötig,
*                          && wenn getrennt compiliert wird

* Prüfung der .DBF- und .NTX-Dateien und ggf. Neuanlage, falls
* nicht vorhanden
* < entsprechende Anweisungen >

auswahl = 0
aktiv = .T.
DO WHILE AKTIV
  DO RAHMEN && Grund-Bildschirm anzeigen

  * Titelbildschirm mit Hauptmenü anzeigen
  *
  * < entsprechende Anweisungen >

  * Funktions-Auswahl einlesen in Variable auswahl
  *
  * < entsprechende Anweisungen >

  DO CASE
    CASE auswahl = 0
      aktiv = .F.
    CASE auswahl = 1
      DO NEUEIN
    CASE auswahl = 2
      DO SUCHEN
    CASE auswahl = 3
      DO AENDERN
    CASE auswahl = 4
      DO LOESCHEN
    CASE auswahl = 5
      DO LISTEN
  ENDCASE
ENDDO
```

```
* Abschließende Programmschritte, wie z.B. Dateien schließen
* und Bildschirm löschen
*
* < entsprechende Anweisungen >

QUIT
```

2.2 Die Gliederung eines Clipper-Programms in Prozeduren

Prozeduren kann man sich als einzelne, in sich abgeschlossene Programm-Bausteine vorstellen, die von anderen Programmteilen aufgerufen werden. An Prozeduren können - falls erforderlich - vom aufrufenden Programm Werte übergeben werden (sogenannte *Parameter*), die dann durch die Prozedur verarbeitet werden. Eine Prozedur ist in der Lage, Werte zu verändern, die bereits im aufrufenden Programmteil bekannt sind, oder, mit anderen Worten ausgedrückt, die Inhalte von Variablen, die bereits im aufrufenden Programmteil definiert wurden, können durch eine Prozedur verändert werden.

Der Aufruf einer Prozedur aus einem Programmteil erfolgt mit der Anweisung `DO <Prozedur-Name> [WITH <Parameterliste>]`. Hierbei ist der Zusatz `WITH` und die Angabe einer Parameterliste nur dann erforderlich, wenn an die Prozedur tatsächlich Werte übergeben werden sollen. Wenn das der Fall ist, muß als erste Anweisung im Anweisungstext der Prozedur die Anweisung `PARAMETERS <Parameterliste>` stehen. Die hier in der `<Parameterliste>` enthaltene Anzahl von Namen muß mit der Anzahl der beim Aufruf der Prozedur angegebenen Parameter übereinstimmen; die Namen der hier angegebenen Variablen müssen jedoch nicht mit den evtl. beim Aufruf angegebenen Namen übereinstimmen.

Nach der Syntax der dBASE-Sprache wird beim Aufruf einer Prozedur mit der Anweisung `DO <Prozedurname>` entweder eine `.PRG`-Datei mit entsprechendem Namen gesucht und ausgeführt oder in der zuletzt mit einer `SET PROCEDURE TO <Prozedurdatei>` angegebenen Prozedurdatei nach einer Zeile `PROCEDURE <Prozedurname>` gesucht. In diesem Fall werden die nachfolgenden Anweisungen bis zu einer `RETURN`-Anweisung ausgeführt. Wichtig ist, daß der Name einer Prozedurdatei *nicht* mit dem Namen einer im Programm aufgerufenen Prozedur übereinstimmen darf. Entweder ist der Name einer Prozedur also durch die Titelzeile `PROCEDURE <Prozedurname>` innerhalb einer Prozedurdatei gekennzeichnet oder die Prozedur steht allein in einer `.PRG`-Datei; in diesem Fall wird die Prozedur durch den Dateinamen benannt und über diesen Namen gefunden.

In Clipper hat dieses Verfahren weiterhin volle Gültigkeit, es gibt jedoch eine Lockerung dieser Vereinbarung. In jedem Teil des Quellcodes, also in jeder .PRG-Datei dürfen Prozedur-Definitionen enthalten sein. Diese Prozeduren müssen jeweils durch eine Titelzeile PROCEDURE <Prozedurname> eingeleitet und mit einer RETURN-Anweisung abgeschlossen werden. Damit ist also bei Clipper die Verwendung einer getrennten Prozedur-Datei und die Anweisung SET PROCEDURE TO <Prozedur-Datei> eigentlich nicht erforderlich. Um kompatibel zu dBASE III zu bleiben, ist die Verwendung von Prozedurdateien jedoch möglich.

Beachten Sie, daß der Name einer .PRG-Datei nicht mit dem Namen einer irgendwo im Programm definierten Prozedur übereinstimmen darf.

Wie bereits gesagt wurde, ist es für die Laufzeit eines compilierten Programms unerheblich, ob das Quellprogramm in mehrere einzelne .PRG-Dateien untergliedert oder ob der gesamte Programmtext in einer großen .PRG-Datei enthalten ist. In der Praxis hat sich jedoch gezeigt, daß die Untergliederung eines Programms in logisch abgegrenzte Teil-Quelldateien sehr vorteilhaft ist. Erstens ist es einfacher und übersichtlicher, eine kleine Datei mit einem Text-Editor zu bearbeiten, zweitens können bestimmte Teilprogramme oft für verschiedene Anwendungen "verwertet" werden, so daß sich umständliches Herauskopieren aus einer großen Datei und das Einfügen in eine andere Datei erübrigt. Drittens erlaubt die Untergliederung das weiter unten beschriebene Compilieren "auf Raten", wodurch Übersetzungszeiten verkürzt werden können.

2.3 Wie ein Clipper-Quellprogramm eingegeben wird

Bei Clipper handelt es sich bei den Quellprogramm-Dateien - wie bei dBASE auch - um reine ASCII-Text-Dateien, so daß zum Erstellen solcher Dateien prinzipiell jeder Text-Editor geeignet ist. Besonders wichtig ist jedoch, daß der Editor keine speziellen Steuerzeichen (z.B. zur Text-Formatierung) einfügt, da diese vom Compiler in den meisten Fällen nicht ignoriert werden, sondern zu einem Übersetzungsfehler führen. Steuerzeichen werden meistens von Textverarbeitungsprogrammen eingefügt. Selbst wenn diese Programme über einen "Programm-Modus" verfügen, kann es zu Fehlern kommen. Diese Eigenschaft weist insbesonders das Textprogramm *Wordstar* 3.4 auf, da hier Sonderzeichen des IBM-Zeichensatzes (also auch die Umlaute) durch eine 3-Byte Zeichensequenz dargestellt werden.

Bestens geeignet zur Erstellung der Quellprogramme ist der "*KRS Text- und Programm-Editor*". Dieser Editor ist sehr schnell, ermöglicht die gleichzeitige Darstellung von bis zu vier Textdateien auf dem Bildschirm und verfügt über viele nützliche Funktionen zur Text-Manipulation. Besonders die vielseitigen Such- und Austauschfunktionen, sowie die automatische Markierung bestimmer Schlüsselworte beim Laden der Textdateien machen ihn zur Bearbeitung von Clipper-Quellprogrammen sehr wertvoll. Der zu bearbeitende Text wird vollständig im Arbeitsspeicher gehalten. Dadurch wird die hohe Arbeitsgeschwindigkeit erreicht. Da jedoch der gesamte, im Rechner frei verfügbare Speicher benutzt wird, lassen sich Dateien von einer Größe bearbeiten, die das "erträgliche Maß" für ein modular aufgebautes Quellprogramm weit überschreiten. Der Editor wird zudem recht preisgünstig auf dem Markt angeboten.

Viele Anwender setzen auch den von IBM erhältlichen Editor PE (Personal Editor) ein, der ebenfalls sehr leistungsfähig ist.

Gut geeignet ist auch der in *Turbo Pascal* von Borland, USA integrierte Text-Editor. Auch dieser Editor ist recht schnell, "kennt" die *Wordstar*-Kommandos und hält den Text im Arbeitsspeicher. Allerdings ist die Speichergröße auf ca. 64 KByte begrenzt. Außerdem verarbeitet der Editor keine Tabulator-Zeichen, wie sie von anderen Editoren in den Text eingefügt werden. Der Editor stellt diese Zeichen als "I" dar, ohne eine Einrückung vorzunehmen.

Natürlich können Sie Programmtexte auch mit dem in dBASE III enthaltenen Editor bearbeiten. Dieser Editor wird durch den Befehl MODIFY COMMAND aufgerufen. Nachteilig ist jedoch, daß wenige Funktionen zur Textbearbeitung vorhanden sind und daß die Textgröße stark eingeschränkt ist. Es kann sogar passieren, daß bei Überschreiten der maximalen Länge Teile des Textes verlorengehen. Außerdem wird der Editor bei größeren Texten - besonders im Einfüge-Modus - sehr langsam.

2.4 Wie ein Clipper-Programm compiliert wird

Im einfachsten Fall ruft man den Clipper-Compiler zusammen mit dem Namen des Hauptmoduls auf. Alle in diesem Modul mit `DO <Prozedurname>` aufgerufenen Prozeduren werden automatisch mitcompiliert. Clipper sucht dazu auf dem aktuellen Laufwerk im aktuellen Verzeichnispfad nach einer Datei `<Prozedurname>.PRG`, öffnet und liest sie, um sie anschließend zu compilieren. Ebenso werden die in einem Untermodul aufgerufenen Prozeduren behandelt. Falls im Quellprogramm eine `SET PROCEDURE TO <Prozedurdatei>`-Anweisung steht, so wird auch diese Datei automatisch verarbeitet. Ebenso sucht und verarbeitet Clipper eventuell im Programm angesprochene Format- und Label-Dateien (`.FMT` und `.LBL`).

Während des Compiler-Laufes erscheinen für jede Datei auf dem Bildschirm Meldungen der Form

`Compiling <Dateiname>` (`<Dateiname>` wird übersetzt)

als Hinweis, welche Quell-Datei gerade bearbeitet wird. Falls Clipper eine Datei nicht finden kann, erscheint folgende Meldung:

`Compiling <Dateiname>`	(`<Dateiname>` wird übersetzt)
`cannot open, assumed external`	(kann nicht geöffnet werden, wird als extern angenommen)

Derartige Hinweise führen nicht zum Abbruch des Übersetzungsvorganges; sie dienen lediglich als Warnung, daß bestimmte Module nicht übersetzt werden konnten.

Wenn die Übersetzung abgeschlossen ist, hat Clipper eine Datei mit dem Zusatz `.OBJ` im aktuellen Verzeichnis auf dem aktuellen Laufwerk angelegt. Wurden während der Übersetzung keine Warnungen über fehlende Dateien ausgegeben, dann enthält diese Objekt-Datei alle nötigen Informationen zur Erzeugung eines lauffähigen Programms mit dem anschließenden Linklauf.

2.5 Compilieren "auf Raten"

Der oben beschriebene Übersetzungsvorgang ist bei kleineren Programmen, die nur aus wenigen Modulen bestehen, sicher am besten geeignet. Wenn Sie jedoch größere Programmsysteme entwickeln, ist es sicher nicht sinnvoll, für jede noch so kleine Änderung in einem Modul alle Module neu zu übersetzen. In diesem Fall sollten Sie die einzelnen Programm-Module getrennt übersetzen.

Ab der Clipper-Version Herbst '86 wurde ein neuer Compiler-Schalter (-m) eingeführt. Wenn Sie diesen angeben, sucht Clipper nicht mehr nach aufgerufenen Untermodulen. Rufen Sie also den Compiler z.B. mit dem Kommando

```
CLIPPER ADRVERW -m
```

auf, um nur die Quelldatei ADRVERW.PRG zu übersetzen.

Wenn Sie die Methode zur getrennten Übersetzung anwenden, müssen Sie natürlich jedes Modul Ihres Programms auf diese Weise mindestens einmal übersetzen. Als Ergebnis erhalten Sie eine entsprechende Anzahl von Objektdateien (.OBJ), die Sie anschließend mit dem Linker zu einem lauffähigen Gesamtprogramm zusammenbinden müssen.

2.6 Das Hilfsprogramm MAKE von Microsoft

Als äußerst nützliche Beigabe erhalten Sie in einigen Softwarepaketen der Firma Mikrosoft (z.B. *MS-C*, Ver. 4.0 oder *MASM*, Ver. 4.0) das Hilfsprogramm MAKE. Dieses Programm überwacht, ob eine Quelldatei geändert wurde und führt in diesem Fall entsprechende Maßnahmen aus. Die Arbeitsweise von MAKE wird in einer vom Anwender zu erstellenden Textdatei beschrieben. Erfreulicherweise arbeitet MAKE ausgezeichnet mit Clipper zusammen. Hier als Beispiel eine MAKE-Textdatei für die weiter oben beschriebene Struktur eines Adressverwaltungsprogramms:

Datei ADRVERW.MAK:

```
ADRVERW.OBJ:          ADRVERW.PRG
   CLIPPER ADRVERW -m

NEUEIN.OBJ:           NEUEIN.PRG
   CLIPPER NEUEIN -m

SUCHEN.OBJ:           SUCHEN.PRG
   CLIPPER SUCHEN -m

AENDERN.OBJ:          AENDERN.PRG
   CLIPPER AENDERN -m

LOESCHEN.OBJ:         LOESCHEN.PRG
   CLIPPER LOESCHEN -m

LISTEN.OBJ:           LISTEN.PRG
   CLIPPER LISTEN -m

ANZEIGE.OBJ:          ANZEIGE.PRG
   CLIPPER ANZEIGE -m

DRUCK.OBJ:            DRUCK.PRG
   CLIPPER DRUCK -m

ETIKETT.OBJ:          ETIKETT.PRG
   CLIPPER ETIKETT -m

HROUT.OBJ:            HROUT.PRG
   CLIPPER HROUT -m

ADRVERW.EXE:          ADRVERW.OBJ NEUEIN.OBJ SUCHEN.OBJ \
                      AENDERN.OBJ LOESCHEN.OBJ LISTEN.OBJ \
                      ANZEIGE.OBJ, DRUCK.OBJ ETIKETT.OBJ \
                      HROUT.OBJ
   PLINK86 @ADRVERW.LNK
```

Diese MAKE-Datei hat folgenden Aufbau:

Am Anfang einer Zeile steht der Name einer *abhängigen Datei*, gefolgt von einem Doppelpunkt. Danach folgt der Name der *Ausgangsdatei*. Die abhängige Datei ist die Datei, die neu erzeugt werden muß, wenn in der Ausgangsdatei Änderungen durchgeführt wurden. Bei Clipper muß immer dann, wenn ein Programm-Modul (also eine .PRG-Datei) geändert wurde, die entsprechende Objekt-Datei (.OBJ) neu gebildet werden. In der nächsten Zeile nach abhängiger und Ausgangsdatei folgen *ohne* Leerzeilen die DOS-Kommandos, die erforderlich sind, um die abhängige Datei neu zu bilden. Hier also der Aufruf des Clipper-Compilers zur Übersetzung eines Einzelmoduls. Eine Leerzeile trennt einen Anweisungsblock vom nächsten.

Im letzten Anweisungsblock sehen Sie, wie die ausführbare Programmdatei (ADRVERW.EXE) als abhängig von allen .OBJ-Dateien gekennzeichnet wird. Sie sehen auch, wie durch das Backslash-Symbol (\) eine Fortsetzung in der nächsten Textzeile gekennzeichnet wird. Dieser Block bewirkt, daß immer dann, wenn sich eine oder mehrere Objekt-Dateien geändert haben, der Linker aktiviert wird, um eine neue .EXE-Datei zu erzeugen. Der Linker (hier PLINK86) wird mit einer Link-Kommando-Datei (ADRVERW.LNK) aufgerufen, in der die erforderlichen Linker-Kommandos enthalten sind.

Die Mühe, die Sie sich einmal machen müssen, um die MAKE-Textdatei zu erstellen, zahlt sich später vielfach aus, da Sie sich bei Änderungen in Quell-Modulen keine Gedanken mehr zu machen brauchen, welche Module neu compiliert werden müssen. Das ist besonders praktisch, wenn Sie beim Programmtest in dem einen oder anderen Modul "Schönheits-Korrekturen" einfügen, wie z.B. Änderung eines Textes oder eines Anzeigeattributes. Solche Änderungen werden dann bei nächster Gelegenheit automatisch mitberücksichtigt.

Sie rufen MAKE einfach aus DOS mit dem Kommando

```
MAKE <MAKE-Dateiname>
```

auf.

Mit dem Clipper Compiler Sommer '87 wird standardmäßig ein MAKE-Hilfsprogramm mitgeliefert, das prinzipiell genauso arbeitet wie das oben beschriebene MAKE von Microsoft. Weitere Informationen finden Sie in der Dokumentation zum Clipper-Compiler.

2.7 Das Linken eines Clipper-Programms

Zusammen mit dem Clipper-Compiler wird ein Linker ausgeliefert (*PLINK86* der Fa. Phoenix, USA). Dieser Linker ermöglichst es, Programmsysteme mit Overlay-Dateien zu erstellen. Die Anwendung von Overlay-Dateien ist im folgenden Kapitel beschrieben.

Die gelieferte Version von *PLINK86* ist eine "Sonderanfertigung" für den Clipper-Compiler, d.h., der Linker arbeitet nur, wenn mindestens ein Objekt-Modul mit dem Clipper-Compiler erzeugt wurde. Diese Einschränkung ist verständlich, da die "normale" Version von *PLINK86* alleine zu etwa dem halben Preis von Clipper verkauft wird. Eine Beschreibung der *PLINK86*-Kommandos finden Sie im Clipper-Handbuch. Achten Sie darauf, daß Sie jeweils die Version von *PLINK86* benutzen, die zusammen mit der aktuellen Clipper-Version ausgeliefert wurde (zu Clipper Sommer '87 gehört die Version *PLINK86 2.24*).

Wenn Sie keine Clipper-Programme mit Overlays entwickeln, können Sie ebensogut den mit dem Betriebssystem Ihres Computers gelieferten Linker *LINK.EXE* verwenden, der von der Firma Microsoft entwickelt wurde.

Neuere Versionen dieses Linkers (ab Version 3.15) arbeiten deutlich schneller als *PLINK86*. Sie erhalten diese Versionen mit den Microsoft-Programmpaketen, die auch das Hilfsprogramm *MAKE* enthalten. (Derzeit aktuell ist Version 5.1, die (wie *PLINK86*) als "Overlay Linker" bezeichnet wird; allerdings können hiermit keine Overlay-Programme für Clipper erzeugt werden, sondern nur solche, die mit Microsoft-Spachen (z.B. *MS-PASCAL*) übersetzt wurden.

Der Aufruf aus der DOS-Kommandozeile erfolgt nach folgendem Schema

```
LINK obj1[+obj2…],[.EXE-Datei],[.MAP-Datei],lib1[+lib2…]
```

Hierbei bezeichnen `objx` die Namen der Objektmodule. `.EXE-Datei` gibt den Namen der zu erstellenden ausführbaren Datei an. Entfällt er, wird der Name des ersten angegebenen Objektmoduls angenommen. `.MAP-Datei` bezeichnet den Namen einer vom Linker erzeugten Textdatei in der die Adress-Referenzen enthalten sind. Entfällt diese Angabe, erhält die Datei den Namen des ersten angegebenen Objekt-Moduls. Will man keine `.MAP`-Datei erstellen, muß als Name das Wort `NUL` angegeben werden. `libx` sind die Namen der abzusuchenden Bibliothek-Dateien, dabei sollte im Normalfall zuerst `CLIPPER.LIB` angegeben werden. Beachten Sie, daß die trennenden Kommata auch dann angegeben werden müssen, wenn optionale Namen entfallen, so z.B.:

```
LINK ADRVERW+PROCS,,NUL,CLIPPER+MEINELIB
```

Bei neueren Versionen des Microsoft-Linkers müssen Sie ggf. einige Schalter angeben.

Wenn z.B. beim Linken die Fehlermeldung

```
too many segments
```

auftritt, müssen Sie eine größere Segmentzahl freigeben. Dazu dient der Schalter `/SEGMENTS:Anzahl` oder `/SE:Anzahl`. Dabei gibt `Anzahl` die freizugebenden Segment an. Werte von 250 bis 300 sind bei Clipper-Anwendungen meistens ausreichend.

Beispiel für den Aufruf:

```
LINK ADRVERW+PROCS,,NUL,CLIPPER+MEINELIB /SE:300
```

Mit dem Schalter `/EXEPACK` erreicht man, daß die erzeugte `.EXE`-Datei komprimiert gespeichert wird. Man kann dadurch Speicherplatz auf der Diskette oder Festplatte einsparen. Der Platzbedarf im Arbeitsspeicher wird jedoch dadurch *nicht* geringer.

Beispiel für den Aufruf:

```
LINK ADRVERW+PROCS,,NUL,CLIPPER+MEINELIB /EXEPACK
```

3 Overlays

Zur Realisierung größerer Programmsysteme bietet Clipper die Aufteilung eines Programms in sogenannte *Overlays*.

Ein "normal" compiliertes und gebundenes Programm ist in einer einzigen Datei (EXE) enthalten, die aus dem Betriebssystem in den Arbeitsspeicher geladen und deren Inhalt dann als Programm ausgeführt wird.

Hierbei ergeben sich natürlich Grenzen für die Programmgröße, die vom vorhandenen (und besonders auch von dem durch das System adressierbaren) Arbeitsspeicher abhängen.

Man darf hierbei nicht nur die Größe der EXE-Datei in Betracht ziehen, sondern man muß berücksichtigen, daß vom Programm zusätzlich Speicher benötigt wird, um Variablen und interne Sprungadressen zu speichern. Weiterer Speicher wird für Dateipuffer benötigt.

Auch das Betriebssystem verbraucht Speicherplatz und unter Umständen wird ein zusätzlicher Teil des Speichers durch residente Hintergrundprogramme benutzt.

Man könnte natürlich ein Anwendungsprogramm in mehrere kleinere Teilprogramme zerlegen, die getrennt compiliert und gelinkt werden. Diese Programme müßten dann jedoch einzeln vom Betriebssystem her gestartet werden.

Da die Grundgröße einer Clipper-Programmdatei mindestens ca. 158 KByte ist, belegt man unnötig viel Platten- oder Diskettenspeicherplatz. Außerdem sind in jedem Modul immer wieder die gleichen Grundroutinen und eventuell gleiche selbst-definierte Funktionen und Prozeduren enthalten.

Daraus ergibt sich, daß eine Aufspaltung in Einzelprogramme in den meisten Fällen nicht praktikabel ist.

Bei der *Overlay-Technik* besteht ein Programm aus einem Grund- oder Haupt-Modul. Dieses Modul wird beim Programmstart vom Betriebssystem in den Arbeitsspeicher geladen und danach ausgeführt. Das Grundmodul verbleibt während der gesamten Laufzeit des Programms im Arbeitsspeicher.

In diesem Grundmodul sind Routinen enthalten, die es ermöglichen, weitere Programmteile in einen dafür freigehaltenen Arbeitsspeicherbereich zu laden, wenn diese Teile benötigt werden. Die Grund-Idee ist, daß diese nachgeladenen

Programmteile sich einen gemeinsamen Speicherbereich teilen, d.h., wenn ein neuer Programmteil benötigt wird, dann wird er in denselben Speicherbereich geladen und er überschreibt oder überlagert (*to overlay - überlagern)* dadurch den dort zuvor geladenen Teil.

Aus diesem Grundkonzept ergeben sich einige wichtige Punkte:

- In einem überlagerten Programmteil dürfen sich keine Routinen befinden, die von einem anderen überlagerten Programmteil benötigt werden, da sich beide Teile nicht gleichzeitig im Arbeitsspeicher befinden. Aus dieser Feststellung ergibt sich, daß gemeinsam genutzte Routinen im Hauptmodul enthalten sein müssen.

- Der für die zu überlagernden Programmteile zu reservierende Speicher muß so groß sein, daß das größte Programmteil hineinpaßt.

- Wenn zwei überlagerte Programmteile Daten austauschen müssen, so ist es erforderlich, daß diese entweder in Dateien oder im Hauptmodul gespeichert werden.

- Eine Overlay-Struktur ist nur dann sinnvoll, wenn sich das Gesamtprogramm in einen Hauptteil und in *mindestens zwei* weitere Teile aufgliedern läßt, die als Overlay-Module nachgeladen werden können.

Eine Erweiterung des Overlay-Konzeptes besteht darin, daß man nicht nur einen, sondern mehrere Overlay-Bereiche schafft. Dadurch ist es z.B. bei zwei Overlay-Bereichen möglich, daß zwei Module vom Hauptprogrammteil nachgeladen werden können, die dann auch gegenseitig auf Routinen und Daten innerhalb der Overlay-Module zugreifen können.

Wenn schon der Aufbau eines Programms mit einem Overlay-Bereich eine aufmerksame Planung und Gliederung des Programms erfordert, so ist es bei mehreren Overlay-Bereichen natürlich um so wichtiger, das Gesamtprogramm sorgfältig zu strukturieren, damit keine Fehler bei der Ausführung auftreten, die dann besonders schwierig zu finden sind.

3.1 Der Overlay-Linker PLINK 86

Zusammen mit dem Clipper-Compiler wird ein Programm-Linker ausgeliefert, der es ermöglicht, Overlay-Strukturen zu bilden. Neuere Versionen des Linkers von MICROSOFT können zwar ebenfalls Overlay-Strukturen bilden, diese sind jedoch derzeit auf Anwendungsprogramme beschränkt, die mit MICROSOFT-Programmiersprachen erstellt wurden.

Zusätzlich zu PLINK86 wird zum Aufbau von Overlay-Strukturen eine Bibliothek-Datei OVERLAY.LIB benötigt, die ebenfalls zum Lieferumfang von Clipper gehört.

In OVERLAY.LIB sind alle Routinen enthalten, die die Verwaltung der Overlay-Module, also z.B. das Reservieren von Arbeitsspeicher-Bereichen und das Nachladen der Module steuern.

Während die Aufteilung des Arbeitsspeichers nach den grundsätzlichen Vorgaben des Programmierers durch PLINK86 selbständig erfolgt, kann die Speicherung der Overlay-Module auf Platte oder Diskette vom Programmierer bestimmt werden:

1. Hauptmodul und alle Overlays in einer EXE-Datei. Hier wird eine einzige, große Datei gebildet, aus der beim Programmstart nur das Hauptmodul in den Arbeitsspeicher geladen wird. Die Overlays werden automatisch dann geladen, wenn sie benötigt werden. Dabei erfolgt das Positionieren des Dateizeigers in der EXE-Datei automatisch.

   Dieses Verfahren bietet sich an, wenn die Massenspeicher-Kapazität ausreicht, um die Gesamtdatei zu speichern, also besonders bei Festplatten. Man hat dadurch den Vorteil, daß von den maximal möglichen offenen Dateien des Betriebssystems nur eine für die Programmdatei benutzt wird.

2. Hauptmodul in einer EXE-Datei und jedes Overlay in einer getrennten OVL-Datei.

   Diese Lösung bietet sich an, wenn eine einzelne Gesamtdatei zu groß ist, um z.B. auf Diskette gespeichert zu werden. Besonders dann, wenn Programme auf Diskette an Endanwender geliefert werden sollen, ist diese Aufteilung sinnvoll. Es besteht dann beim Endanwender natürlich trotzdem die Möglichkeit, alle Module auf eine Festplatte zu übertragen und von dort zu starten.

Auch dann, wenn in einem System nur Diskettenlaufwerke zur Verfügung stehen, kann man ein Programm so aufteilen. Die internen Overlay-Routinen suchen automatisch alle vorhandenen Laufwerke und Pfade nach den Overlay-Dateien ab. Wird eine Datei nicht gefunden, erzeugt das Programm eine Meldung, die zum Wechsel der Diskette auffordert.

3. Beliebige Kombinationen der Möglichkeiten 1 und 2. So können z.B. bestimmte Overlay-Module zusammen mit dem Hauptmodul in einer Datei enthalten sein und auch bestimmte Overlay-Module in einer Datei kombiniert sein.

   Solche Kombinationen können benutzt werden, um den verfügbaren Speicherplatz auf Disketten optimal auszunutzen, also um z.B. EXE- oder OVL-Dateien mit einer Größe von jeweils ca. 360 KByte zu erzeugen.

Wichtig ist, daß die vorstehend beschriebenen Aufteilungsmöglichkeiten *keinen* Einfluß auf die Aufteilung im Arbeitsspeicher haben; diese wird durch die Gliederung der Programme selbst und durch Linker-Kommandos definiert.

3.2 Beispielprogramm für eine Overlay-Struktur

Das folgende Programm ist unübertroffen einfach und in der Praxis sicher ohne jeden Nutzen, aber für die folgende Beschreibung der Linkerkommandos völlig ausreichend.

```
****
* HAUPT.PRG
*
CLEAR
? "Ich bin das Hauptmodul"
INKEY(0)
DO mod1
DO mod2
DO mod3
? "Das Hauptmodul sagt auf Wiedersehen"
QUIT
*
* Ende von HAUPT.PRG
*
```

```
****
* MOD1.PRG
*
? "Ich bin MOD1"
INKEY(0)
RETURN

* Ende von MOD1

****
* MOD2.PRG
*
? "Ich bin MOD2"
INKEY(0)
RETURN

* Ende von MOD2.PRG

****
* MOD3.PRG
*
? "Ich bin MOD3"
INKEY(0)
RETURN

* Ende von MOD3.PRG
```

Diese vier Programmdateien *müssen getrennt* compiliert werden:

```
CLIPPER HAUPT -m
CLIPPER MOD1
CLIPPER MOD2
CLIPPER MOD3
```

Nun liegen die vier Objekt-Dateien

```
HAUPT.OBJ
MOD1.OBJ
MOD2.OBJ
MOD3.OBJ
```

vor.

Es folgen nun einige Beispiele, wie PLINK86 aufgerufen werden kann, um diese Module in eine Overlay-Struktur einzubinden.

1. Bilden einer einzelnen EXE-Datei:

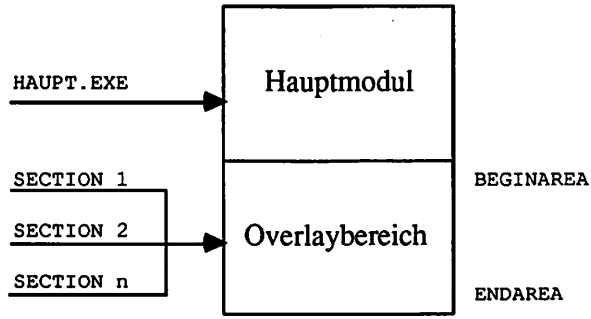
Kommandofolge:

```
PLINK86
FILE HAUPT
BEGINAREA
SECTION FILE MOD1
SECTION FILE MOD2
SECTION FILE MOD3
ENDAREA
;
```

Als Ergebnis erhalten Sie eine Datei HAUPT.EXE, in der das Hauptmodul und alle Overlays enthalten sind.

Die Kommandos BEGINAREA und ENDAREA signalisieren dem Linker, daß ein Overlay-Bereich anzulegen ist, in den wechselweise die Module MOD1, MOD2 und MOD3 zu laden sind. Da keine Angaben über Ausgabedateien enthalten sind, wird der geamte Code in eine EXE-Datei geschrieben, die denselben Namen hat, wie das Hauptmodul.

Beim Autor ergab sich eine EXE-Datei von 161568 Bytes.



Struktur mit einem Overlay-Bereich, in den abwechselnd mehrere Module geladen werden

2. Bilden getrennter Haupt- und Overlay-Dateien:

Kommandofolge:

```
PLINK86
FILE HAUPT
BEGINAREA
SECTION INTO MOD1 FILE MOD1
SECTION INTO MOD2 FILE MOD2
SECTION INTO MOD3 FILE MOD3
ENDAREA
;
```

Wie im ersten Beispiel kennzeichnen die Anweisungen BEGINAREA und ENDAREA, daß ein gemeinsamer Overlay-Bereich gebildet werden soll.

Die Anweisung SECTION gibt jeweils an, welche OBJ-Module zu einem Block zusammengefaßt werden soll, der in den Overlay-Bereich geladen wird. In diesem Beispiel wird jedes OBJ-Modul einzeln in den Overlay-Bereich geladen.

Der Zusatz INTO <Dateiname> gibt an, daß diese Overlay-Module in OVL-Dateien mit den angegebenen Namen geschrieben werden sollen.

Als Ergebnis erhalten Sie eine Datei HAUPT.EXE und die drei Overlay-Dateien MOD1.OVL, MOD2.OVL und MOD3.OVL.

Dabei ergaben sich folgende Dateigrößen:

HAUPT.EXE:	161312	Bytes
MOD1.OVL:	96	Bytes
MOD2.OVL:	96	Bytes
MOD3.OVL:	96	Bytes
gesamt:	161600	Bytes

3. Bilden einer Haupt-Datei und von Overlay-Dateien, die mehrere OBJ-Module enthalten:

```
PLINK86
FILE HAUPT
BEGINAREA
SECTION INTO MOD12 FILE MOD1, MOD2
SECTION INTO MOD3 FILE MOD3
END
;
```

In diesem Fall erhalten Sie folgende Dateien:

HAUPT.EXE:	161408	Bytes
MOD12.OVL:	176	Bytes
MOD3.OVL:	96	Bytes
gesamt:	161680	Bytes

Die Aufteilungsmöglichkeiten sind mit diesen Beispielen natürlich noch lange nicht erschöpft. Es stellt sich aber hier die Frage, warum PLINK86 diese vielfältigen Möglichkeiten bietet.

Hierzu ändern wir zunächst das Programm MOD2.PRG um:

```
****
* MOD2.PRG
*
? "Ich bin MOD2"
INKEY(0)
DO MOD3
RETURN

* Ende von MOD2.PRG
```

Dieses Modul ruft nun seinerseits MOD3 auf. Übersetzen Sie es zunächst neu:

```
CLIPPER MOD2 -m
```

Wenn Sie die Module nun wieder nach dem Verfahren 2 binden, also getrennte Haupt- und Overlay-Dateien für jedes Modul bilden, so erhalten Sie bei der Programm-Ausführung (mit viel Glück) einen Laufzeit-Fehler oder das Programm "hängt sich auf".

Der Fehler rührt daher, daß MOD2 intern MOD3 aufruft. Da jedoch alle Module in getrennten SECTION-Anweisungen angegeben wurden, wird auch nur immer eines der Module in den Overlay-Bereich geladen, so daß sich MOD2 und MOD3 nicht gleichzeitig im Speicher befinden können, obwohl das für die Programmausführung erforderlich wäre, da ja MOD3 von MOD2 aufgerufen wird.

Wenn Sie das Programm ausführen, erkennen Sie zwar an der Meldung "Ich bin MOD3", daß offensichtlich MOD3 geladen wurde, aber anschließend "stürzt" das Programm ab, da der Rücksprung in MOD2 nicht ausgeführt werden kann.

Wenn Sie folgende Link-Methode anwenden, arbeitet das Programm korrekt:

```
PLINK86
FILE HAUPT
BEGINAREA
SECTION INTO MOD1 FILE MOD1
SECTION INTO MOD23 FILE MOD2, MOD3
ENDAREA
;
```

Nun sind die Module MOD2 und MOD3 in einer Sektion zusammengefaßt und werden gemeinsam in den Overlay-Bereich geladen. Natürlich sind auch alle anderen Kombinationen zulässig, die sicherstellen, daß beide Module gleichzeitig im Arbeitsspeicher enthalten sind.

3.3 Strukturen mit mehreren Overlay-Bereichen

Wie Sie dem letzten Beispiel entnehmen konnten, ist es wichtig, daß Module, die sich gegenseitig aufrufen, gleichzeitig im Arbeitsspeicher präsent sind. Im vorangehenden Beispiel wurde das durch eine geeignete SECTION-Anweisung erreicht. Diese Lösung kann immer dann benutzt werden, wenn der zur Verfügung stehende Arbeitsspeicher ausreichend groß ist.

Es gibt jedoch sehr oft Fälle, daß von einem Modul, z.B. MOD1 zunächst ein Modul MOD2 und später ein Modul MOD3 aufgerufen wird. In diesem Fall müßten

alle drei Module gleichzeitig im Arbeitsspeicher enthalten sein, obwohl beim Aufruf von MOD2 die Existenz vom MOD3 nicht erforderlich ist und umgekehrt.

Hier zunächst das modifizierte Modul1 des Beispielprogramms:

```
****
* MOD1.PRG
*
? "Ich bin MOD1"
INKEY(0)
DO MOD2
DO MOD3
RETURN

* Ende von MOD1
```

Compilieren Sie das Modul mit der Anweisung

```
CLIPPER MOD1 -m
```

Vergessen Sie nicht, in MOD2.PRG die eventuell für den "Crash-Test" gemachte Änderung wieder rückgängig zu machen und ggf. auch dieses Modul neu zu compilieren.

Mit einem Overlay-Bereich müssen Sie das Programm wie folgt linken:

```
PLINK86
FILE HAUPT
BEGINAREA
SECTION INTO MOD123 FILE MOD1, MOD2, MOD3
ENDAREA
;
```

Damit erreichen Sie, daß alle drei Module gemeinsam in den Overlay-Bereich geladen werden (im Normalfall würde man ein solches Programm natürlich nicht als Overlay-System aufbauen, da keine Einsparung an Speicherplatz gewonnen wird).

Wenn man ein System mit zwei Overlay-Bereichen aufbaut, dann kann im ersten Bereich MOD1 enthalten sein und im zweiten Bereich wird entsprechend den Aufrufen von MOD1 bzw. von HAUPT das Modul MOD2 oder MOD3 nachgeladen.

Die Linker-Anweisungen lauten:

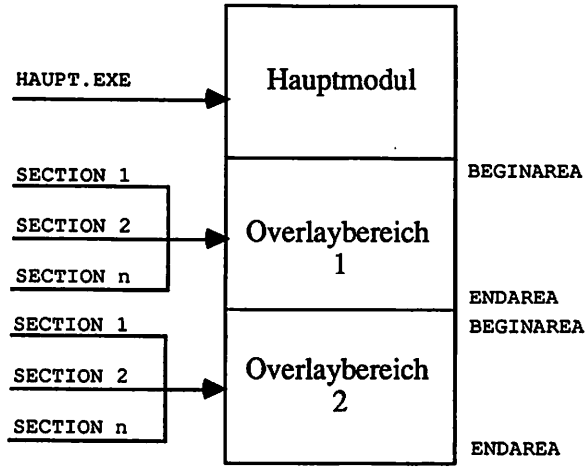
```
FILE HAUPT
BEGINAREA
SECTION INTO MOD1 FILE MOD1
ENDAREA
BEGINAREA
SECTION INTO MOD2 FILE MOD2
SECTION INTO MOD3 FILE MOD3
ENDAREA
;
```

Durch diese Anweisungsfolge wird der Linker aufgefordet, eine Programmstruktur mit zwei Overlay-Bereichen anzulegen. Sie erkennen das an den beiden BEGINAREA...ENDAREA-Blöcken. Jeder dieser Blöcke kennzeichnet einen Overlay-Bereich.

Der erste Bereich wird nur vom Modul MOD1 verwendet. (In einer praktischen Anwendung ist dies natürlich unsinnig, da ja dann kein Overlay-Bereich erforderlich ist. Stattdessen würde man das Modul mit zum residenten Hauptprogramm binden.)

Den zweiten Overlay-Bereich teilen sich die Module MOD2 und MOD3.

Natürlich sind auch bei mehreren Overlay-Bereichen verschiedene Aufteilungen und Zusammenfassungen von Modulen möglich. Wichtig ist, daß immer sichergestellt wird, daß alle Programmteile, die von anderen Programmteilen aufgerufen werden, so in den Speicher geladen werden können, daß dabei der aufrufende Programmteil *nicht überschrieben* wird, da sonst ein Rücksprung nicht ordnungsgemäß erfolgen kann.



Struktur mit zwei Overlay-Bereichen

3.4 Zusammenfassung der PLINK86-Anweisungen zur Overlay-Struktur

`BEGINAREA...ENDAREA`

Dieser Anweisungsblock bewirkt, daß ein Overlay-Bereich im Arbeitsspeicher eingerichtet wird, in den alle Module geladen und in dem sie ausgeführt werden, die zwischen den Schlüsselworten `BEGINAREA` und `ENDAREA` stehen.

`SECTION FILE`

Alle hinter dem Wort `FILE` angegebenen Objekt-Modulnamen werden zu einer Sektion zusammengefaßt, das heißt, daß diese Module immer *gemeinsam* in den Overlay-Bereich geladen werden.

Innerhalb eines `BEGINAREA...ENDAREA`-Blockes sind normalerweise mehrere `SECTION FILE`-Anweisungen enthalten. Es wird immer nur *eine* der Sektionen mit allen darin zusammengefaßten Objekt-Modulen in den Overlay-Bereich geladen.

```
INTO <Dateiname>
```

Dieser Zusatz hinter einer SECTION-Anweisung bestimmt, in welche Datei diese Sektion gespeichert werden soll. Sie hat keinen Einfluß auf die Aufteilung der Overlay-Bereiche oder auf die Zuordnung von Objekt-Modulen zu Sektionen.

Es ist innerhalb eines Link-Laufes erlaubt, bei verschiedenen INTO-Anweisungen denselben Dateinamen anzugeben. Dadurch können verschiedene Sektionen in einer Datei zusammengefaßt werden. Aus dieser Datei werden dann bei der Ausführung die jeweils benötigten Teile nachgeladen.

So erzeugen die folgenden Linker-Anweisungen eine Overlay-Struktur mit zwei Overlay-Bereichen, wie unter 3.3 beschrieben, es wird jedoch nur eine Overlay-Datei MOD123.OVL angelegt:

```
PLINK86
FILE HAUPT
BEGINAREA
SECTION INTO MOD123 FILE MOD1
ENDAREA
BEGINAREA
SECTION INTO MOD123 FILE MOD2
SECTION INTO MOD123 FILE MOD3
ENDAREA
;
```

3.5 Planung eines Clipper-Programms für die Verwendung mit Overlays

Wenn Sie beabsichtigen, in einer Anwendung Overlays einzusetzen, so sollten Sie darauf achten, daß bereits bei der Anlage der Programmstruktur die spätere Aufteilung in Overlays berücksichtigt wird.

Clipper führt aufgrund seiner Sprache mit der Gliederung in Funktionen und Prozeduren ohnehin zu wohlstrukturierten Programmen, die eine Voraussetzung für die Verwendung von Overlays sind.

Das nachstehende Gliederungsschema ist übrigens nicht nur bei der Planung von Overlay-Strukturen wichtig, sondern es führt auch bei "normalen" Programmen zu einer sauberen Aufteilung, so daß Sie sich eigentlich immer daran halten sollten. Darüberhinaus besteht dann die Möglichkeit, eine zunächst nicht mit Overlays geplante Anwendung später leicht so umzuändern, daß sie als Overlay-Struktur ablaufen kann.

1. Hauptmodul

In diesen Programmteil sollten Sie alle Anweisungen, Initialisierungen, Prozedur- und Funktionsdefinitionen aufnehmen, die später von fast allen anderen Programmteilen verwendet werden.

In diesem Teil sollte auch die Haupt-Funktionsauswahl, also meistens eine Gruppe von Menü-Anweisungen enthalten sein, mit denen der Aufruf der verschiedenen Untermodule gesteuert wird.

Es ist fast selbstverständlich, daß im Hauptmodul auch der Programm-Ausstieg, ggf. mit dem Schließen von Dateien enthalten sein sollte.

Dieses Hauptmodul (das durchaus aus mehreren einzelnen PRG-Dateien gebildet werden kann) wird auch in der Overlay-Struktur zum residenten Hauptmodul.

2. Untermodule

Hier sollten Sie zunächst solche Module festlegen, die vom Hauptmodul direkt aufgerufen werden. Es dürfen dabei in dieser Gruppe nur solche Module enthalten sein, die alternativ aufgerufen werden. Auch darf keines der Module ein anderes Modul dieser Gruppe aufrufen.

In der Regel enthalten solche Untermodule die einzelnen Teilabläufe einer Anwendung.

Da jeweils nur eines dieser Module ausgeführt und von zentraler Stelle (dem Hauptmodul) aufgerufen wird, können sich all diese Module einen Overlay-Bereich teilen. Sie werden also beim Linken innerhalb eines BEGINAREA... ENDAREA-Blockes mit separaten SECTION-Anweisungen angegeben.

3. Hilfsmodule

In dieser Gruppe sollten alle Prozeduren und Funktionen zusammengefaßt werden, die von einigen der Untermodule und evtl. vom Hauptmodul aus aufgerufen werden.

Es muß nun entschieden werden, ob diese Hilfsmodule in einem weiteren Overlay-Bereich ablaufen sollen, oder ob sie evtl. mit an das Hauptmodul gebunden werden.

Ein zusätzlicher Overlay-Bereich ist nur dann sinnvoll, wenn man die Hilfsmodule in mindestens zwei unabhängige Gruppen aufteilen kann. Es sollten außerdem in einer Gruppe solche Module zusammengefaßt werden, die öfters hintereinander von anderen Programmteilen aufgerufen werden, um ein ständiges Nachladen zu vermeiden.

4 Die Größe von Clipper-compilierten Programmen

Selbst ein noch so kurzes Quellprogramm, z.B. der "Einzeiler"

```
? "hallo"
```

erzeugt eine ausführbare .EXE-Programmdatei von ca. 158 KByte Größe. Das mag zunächst verwundern und unverständlich erscheinen. Die enorme Codegröße hat jedoch mehrere Gründe, die an der dem Clipper-Compiler zugrundeliegenden Sprachsyntax von dBase III und der dBase-Dateistruktur liegen.

Gemäß dBase III Syntax sind folgende Anweisungen in einem Programm erlaubt:

```
cmd = SPACE(20)
@ 10,10 GET cmd
READ
@ 12,10 SAY &cmd
```

Wenn wir annehmen, daß der Benutzer bei der Ausführung der GET-Anweisung den Text

`log(10)` eingibt, wird über die Makro-Funktion die Anweisung

```
@ 12,10 SAY &cmd
```

in die Anweisung

```
@ 12,10 SAY log(10)
```

umgewandelt. Das bedeutet, daß während der Programmlaufzeit der Logarithmus von 10 berechnet werden muß, d.h. im ausführbaren Programm muß die Funktion `log()` zur Verfügung stehen, obwohl sie im Quellprogramm an keiner Stelle direkt aufgerufen wird. Aus diesem Grund muß Clipper "auf Verdacht" aus der Clipper-Bibliothek alle Funktionen anfordern und diese werden durch den Linker in die .EXE-Datei eingebunden.

Eine ähnliche Situation ergibt sich, wenn in einem Programm Index-Dateien benutzt werden sollen. Der Schlüsselausdruck steht im Kopfsatz der Index-Datei. Er könnte z.B.

```
UPPER(SUBSTR(name,1,5))
```

lauten. Während der Laufzeit muß das Programm in der Lage sein, die Funktionen UPPER() und SUBSTR() auszuführen, obwohl im Programm selbst u.U. an keiner Stelle eine dieser Funktionen aufgerufen wird.

Auch folgende Anweisung ist völlig legal:

```
var = " "
@ 10,10 GET var
READ
&var = 47
```

Benutzereingabe: x

Hier wird während der Programmlaufzeit, bedingt durch die Benutzereingabe, über die Makrofunktion eine Variable x definiert und ihr der Wert 47 zugewiesen. Hier konnte der Compiler während der Programmübersetzung nicht erkennen, daß eine Variable x anzulegen ist und welchen Typ diese Variable hat.

Eine ähnliche Situation tritt auf, wenn in einem Programm eine Datei geöffnet wird und die Feld-Inhalte bearbeitet werden sollen. Die Struktur der Datei, die Anzahl, Namen und Typen der Felder sind während der Programmübersetzung nicht bekannt, da diese Informationen im Kopfsatz der .DBF-Datei stehen.

Aus diesen Gründen muß Clipper in jedes Programm ein Modul einbinden, das eine dynamische Variablenverwaltung ausführt. Dadurch wird zusätzlicher Speicherplatz benötigt.

In den oben erwähnten Bereichen arbeitet ein Clipper-compiliertes Programm immer noch ähnlich wie ein Interpreter.

Compiler für andere Programmiersprachen haben diese Schwierigkeiten nicht, da üblicherweise alle Funktionen im Quellprogramm entweder definiert oder als extern deklariert werden. Ebenso müssen alle Variablen mit Typ und Namen in einem Deklarationsteil angegeben werden.

Mittlerweile werden andere dBase III Compiler angeboten, die angeblich mit wesentlich geringeren .EXE-Dateigrößen auskommen. Bei genauem Studium der Dokumentation erkennt man dann aber, daß zusätzlich vor der Ausführung eines compilierten Programms ein speicher-residentes Modul in den Arbeitsspei-

cher geladen werden muß, das die entsprechenden Laufzeit-Routinen enthält. Somit ist auch bei diesen Programmen ein ähnlich großer Speicherplatz erforderlich.

Auch der dBase III-Interpreter benötigt einen enormen Speicherplatz; er ist sogar in mehrere Overlays gegliedert, die bei Bedarf nachgeladen werden, damit in den üblichen Arbeitsspeichergrößen eine Programmausführung überhaupt möglich ist.

5 Prozeduren

Für dBase-Programmierer sind Prozeduren "alte Bekannte", stellen sie doch in dBase die einzige Möglichkeit dar, ein Programm in Untermodule zu gliedern. Clipper hingegen bietet zusätzlich anwender-definierbare Funktionen, die in den folgenden Kapiteln vorgestellt werden.

Eine Prozedur kann man mit einem Unterprogramm vergleichen. Sie wird aus einem übergeordneten Programm aufgerufen und nach Bearbeitung der Prozedur kehrt der Programmablauf in das aufrufende Programm zurück. Das aufrufende Programm wird dann mit der Anweisung fortgesetztm die auf die Aufruf-Anweisung folgt.

Prozeduren können also immer dann sinnvoll eingesetzt werden, wenn bestimmte Anweisungefolgen an mehreren Stellen in einem Programm wiederholt werden sollen. Statt diese Anweisungen jeweils neu zu schreiben, ersetzt man sie durch einen Prozedur-Aufruf und schreibt die Anweisungen einmal in die Prozedur.

Neben der "klassischen" Verwendung einer Prozedur als Unterprogramm haben Prozeduren in dBase- und Clipper-Programmen jedoch die wichtige Aufgabe, ein Programm in kleinere, überschaubare Abschnitte zu gliedern. So kann man die in Kapitel 2 gezeigte Gliederung eines Programms in mehere Teilmodule dadurch erreichen, daß man jedes Modul in einer Prozedur zusammenfaßt.

Der Aufruf einer Prozedur erfolgt mit der Anweisung

```
DO <Prozedurname> [WITH <Parameterliste>]
```

Das Schlüsselwort `DO` bewirkt den Aufruf der Prozedur, deren Name nach `DO` angegeben wurde. Optional kann das Schlüsselwort `WITH` zusammen mit einem oder mehreren Parametern angegeben werden, die dann durch Kommata getrennt sein müssen.

Parameter sind Werte, die an die Prozedur übergeben werden. Als Parameter sind Variablen, Ausdrücke oder Konstanten erlaubt.

Die Struktur einer Prozedur entspricht stets dem folgenden Schema:

```
PROCEDURE <Prozedurname>
[PARAMETERS <Parameterliste>]
*
* Anweisungen
*
RETURN
```

Hinter dem Schlüsselwort PROCEDURE wird der Name der Prozedur angegeben, mit dem sie von anderen Programmteilen aufgerufen werden kann.

Die Zeile PARAMETERS <Parameterliste> ist optional; sie muß dann angegeben werden, wenn an die Prozedur Werte übergeben werden sollen. Die Parameterliste muß einen oder mehrere, durch Komma getrennte Variablennamen enthalten. Diese Variablen dienen als "Auffangbehälter" für die beim Aufruf der Prozedur in der dort angegebenen Parameterliste angegebenen Werte.

Über die Namen dieser Variablen können dann innerhalb der Prozedur die übergebenen Werte "abgeholt" werden.

Entsprechend der dBase III-Syntax gibt es auch bei Clipper zwei Ausnahmefälle für die Standard-Struktur einer Prozedur.

Die Zeile PROCEDURE <Prozedurname> kann entfallen, wenn die Prozedur unmittelbar am Anfang einer Quellprogramm-Datei steht, deren Name nach dem Schema <Prozedurname>.PRG aufgebaut ist.

Die Zeile RETURN kann entfallen, wenn danach keine weiteren Anweisungen mehr in der Quellprogramm-Datei stehen, oder wenn mit der nächsten Anweisung die Definition einer neuen Prozedur (Schlüsselwort PROCEDURE) oder einer Funktion (Schlüsselwort FUNCTION) beginnt.

In "klassischen" dBase-Programmen ist die Regel, daß jede Prozedur in einer separaten .PRG-Datei gespeichert wird, so daß dort diese beiden Ausnahmenfälle zu "Regelfällen" werden. Anders ist es in einer Prozedur-Datei, die als "Sammelbecken" für verschiedene Prozeduren dient.

In Clipper ist diese Aufteilung in einzelne .PRG-Dateien ohne Einschränkung möglich, man wird jedoch (sofern man die Programme nicht auch unter dBase III ablaufen lassen will) meistens verschiedene, logisch zusammengehörige Prozeduren in einer .PRG-Datei zusammenfassen.

Wichtig ist dabei, daß entweder bei der ersten Prozedur die Zeile PROCEDURE <Prozedurname> entfällt (die Prozedur wird dann mit dem Dateinamen aufgerufen) oder daß der .PRG-Dateiname *nicht* mit dem Namen einer darin enthaltenen Prozedur übereinstimmt.

Spezielle Prozedur-Dateien, wie man sie bei dBase III kennt (sie werden mit SET PROCEDURE TO <Dateiname> aktiviert), sind bei Clipper nicht erforderlich, da in Clipper-Programmen in jeder .PRG-Datei Prozedur-Definitionen enthalten sein dürfen. Clipper erkennt und bearbeitet jedoch die SET PROCEDURE-Anweisung richtig, um kompatibel zu dBase III zu sein.

5.1 Werte-Übergabe bei Prozeduren, Gültigkeit von Variablen

Das folgende Beispielprogramm soll zur Erläuterung der Werte-Übergabe und der Gültigkeit von Variablen dienen:

```
CLEAR SCREEN
PUBLIC var1, var2
var2 = 123
a = "A"
b = "B"
c = "C"
d = "D"
? "Im Hauptprogramm"
? a
? b
? c
? d
DO test WITH a, b + ""
? "Wieder im Hauptprogramm"
? a
? b
? c
? d
? var1
? var2
? var3
QUIT
```

```
PROCEDURE test
PARAMETERS x, y
PRIVATE d
x = "Das ist x"
y = "Das ist y"
c = "Das ist c"
d = "Das ist d"
var1 = "Das ist var1"
var2 = "Das ist var2"
var3 = "Das ist var3"
? "In der Prozedur"
? x
? y
? a
? b
? c
? d
? var1
? var2
? var3
RETURN
```

Das Programm erzeugt folgende Ausgabe:

```
Im Hauptprogramm
A
B
C
D
.F.
123
In der Prozedur
Das ist x
Das ist y
Das ist x
B
Das ist c
Das ist d
Das ist var1
Das ist var2
Das ist var3
```

```
Wieder im Hauptprogramm
Das ist x
B
Das ist c
D
Das ist var1
Das ist var2
```

Danach bricht das Programm mit der Fehlermeldung

```
undefined Identifier VAR3
```

ab.

Wir wollen nun das "Leben" der verschiedenen Variablen betrachten:

Variable `a`:

Die Variable `a` wird als Parameter an die Prozedur "per Referenz" übergeben. "Per Referenz" bedeutet, daß an die Prozedur nicht der Inhalt, der Variablen, sondern ihre Adresse übergeben wird, oder genauer die Adresse des Speicherplatzes, in dem der Inhalt der Variablen abgeglegt ist.

Die in der Prozedur angegebene "Auffang-Variable" `x` erhält nun ebenfalls die Adresse des in `a` gespeicherten Wertes zugewiesen.

Wenn jetzt in der Prozedur mit der Anweisung `x = "Das ist x"` eine neue Wertzuweisung erfolgt, bedeutet das, daß die Variable `x` und damit auch die Variable `a` auf den Speicherplatz dieses neuen Wertes verweisen.

Damit ergeben die Anweisungen `? x` und `? a` in der Prozedur jeweils die Ausgabe `Das ist x`. Auch im Hauptprogramm erzeugt die zweite Anweisung `? a` nun die Ausgabe `Das ist x`.

Der Inhalt der Variablen a wurde also durch die Prozedur verändert.

Variable `b`:

Die Variable `b` wird an die Prozedur nicht direkt als Parameter übergeben; stattdessen steht `b` in einem Ausdruck: `b + ""`. Die Übergabe an die Prozedur erfolgt nun "als Wert". Das bedeutet, daß in diesem Fall nicht die Adresse eines Speicherplatzes mit einem Wert, sondern der Wert selbst übergeben wird. Etwas

anderes ist auch nicht möglich, da das Ergebnis des Ausdrucks `b + ""` ja nicht in einem Speicherplatz enthalten ist, sondern erst bei Aufruf der Prozedur berechnet wird.

Dieser Wert wird in die prozedur-interne Variable `y` übertragen. Anschließend erhält `y` den neuen Wert `Das ist y`. Die nachfolgenden Ausgabe-Anweisungen `? b` in der Prozedur und im Hauptprogramm beweisen, daß in diesem Fall der Inhalt der Variablen `b` nicht verändert wurde.

Variable `c`:

Diese Variable wurde *nicht* als Parameter an die Prozedur übergeben. Dennoch hat die Prozedur mit der Anweisung `c = "Das ist c"` ihren Wert verändert, die die entsprechenden Ausgabeanweisungen zeigen.

Eine Variable, die in einem übergeordneten Programmteil definiert wurde, ist auch in den untergeordneten Programmteilen definiert und kann von diesen geändert werden.

Variable `d`:

Die Variable `d` erhält im Hauptprogramm den Wert "`D`" und sie hat diesen Wert auch noch nach Bearbeitung der Prozedur, obwohl innerhalb der Prozedur die Anweisung `d = "Das ist d"` steht. Durch die Anweisung `PRIVATE d` wird innerhalb der Prozedur eine "Privat-Variable" `d` deklariert, die nichts mit der namensgleichen Variablen im Hauptprogramm zu gemeinsam hat. Die Prozedur kann ihr einen Wert zuweisen (wie es ja auch in diesem Beispiel erfolgt), jedoch beeinflußt das nicht den Inhalt von `d` im Hauptprogramm.

Variable `var1`:

Diese Variable wurde im Hauptptogramm als `PUBLIC` deklariert. Das bedeutet, daß die im Hauptprogramm und in allen untergeordneten Programm-Ebenen bekannt ist. Die erste Ausgabe-Anweisung im Hauptprogramm `? var1` erzeugt `.F.` als Ergebnis. Per Definition in der dBase-Sprache haben alle `PUBLIC`-Variablen solange den Typ logisch und enthalten den Wert `.F.`, bis ihnen ein anderer Wert zugewiesen wird.

Eine Ausnahme bildet in Clipper die spezielle Variable `CLIPPER`. Wird diese in einem Programm als `PUBLIC` deklariert, erhält die den Wert `.T.`; damit kann man in einem Programm prüfen, ob es unter Clipper oder dBase III abläuft, da in dBase III diese Variable ja per Definition den Wert `.F.` annimmt.

Die Prozedur weist `var1` den String-Wert "`Das ist var1`" zu; dieser Wert ist anschließend auch im Hauptprogramm gültig.

Variable `var2`:

Wie `var1` wird auch diese Variable im Hauptprogramm als `PUBLIC` deklariert. Sie erhält hier den Wert `123`, also einen numerischen Wert zugewiesen. In der Prozedur erfolgt die Zuweisung `var2 = "Das ist var2"`. Dieser neue Wert ist anschließend auch im Hauptprogramm gültig. Man erkennt daran, daß eine Prozedur an eine `PUBLIC`-Variable nicht nur einen neuen Wert zuweisen, sondern auch den Typ der Variablen ändern kann.

Variable `var3`:

Diese Variable wird erst in der Prozedur definiert und dort enthält die den Text "`Das ist var3`". Beim anschließenden Versuch, diese Variable im Hauptprogramm auszugeben, bricht das Programm mit einem Laufzeitfehler ab, der die Variable als nicht definiert meldet.

Variablen, die innerhalb einer Prozedur definiert werden, sind lokal, d.h. in höheren Programm-Ebenen nicht "sichtbar".

An diesen Fällen können Sie erkennen, wie vielfältig die Möglichkeiten sind, mit der eine Prozedur Werte verändern kann. Das mag zwar in einigen Situationen vorteilhaft sein, in anderen Fällen kann es jedoch zu schwer zu lokalisierenden Fehlern führen.

Sie sollten bei Prozeduren immer dann eine `PRIVATE`-Anweisung einfügen, wenn Sie innerhalb der Prozedur Variablen verwenden möchten und wenn Sie ausschließen wollen, daß dadurch Werte zufällig namensgleicher Variablen in anderen Programmteilen verändert werden.

Wie aus dem Beispiel ersichtlich, sind zwar innerhalb von Prozeduren definierte Variablen prinzipiell lokal, jedoch kann nie ausgeschlossen werden, daß in einer höheren Programm-Ebene eine gleichnamige Variable verwendet wird. Aus diesem Grund ist die `PRIVATE`-Deklaration in jedem Fall sinnvoll, wenn man universell verwendbare Prozeduren schreiben will.

Wenn Sie eine Variable als Parameter an eine Prozedur übergeben müssen und gleichzeitig verhindern wollen, daß die Prozedur den Inhalt der Variablen verändern kann, sollten Sie die Variable nicht direkt, sondern in einem Ausdruck

innerhalb der Parameterliste hinter `WITH` angeben. Neben der oben gezeigten Möglichkeit `b + ""` können Sie eine Variable auch einfach in runde Klammern setzen `(b)`, um einen Ausdruck zu "simulieren".

Wenn Sie innerhalb einer Prozedur gezielt die Inhalte von Variablen im aufrufenden Programm ändern wollen, sollten Sie diese Variablen als Parameter an die Prozedur übergeben (wie `a` im obigen Beispiel) und *nicht* direkt einen Variableninhalt ändern (wie bei `c`). Dadurch erreichen Sie, daß die Prozedur unverändert auch in anderen Programmen verwendet werden kann, ohne daß Sie evtl. anders benannte Variablen berücksichtigen müssen.

Wichtig ist in jedem Fall eine entsprechende Dokumentation (sinnvoll innerhalb von Kommentarzeilen am Anfang der Prozedur-Definition), die aussagt, welchen Einfluß die Prozedur auf außerhalb definierte Variablen hat.

Wenn Sie die beiden letzten Punkte mißachten, bauen Sie sich und anderen (die verzweifelt herauszufinden suchen, die Sie das Programm zum Arbeiten gebracht haben) ausgezeichente Stolperfallen in ein Programm ein.

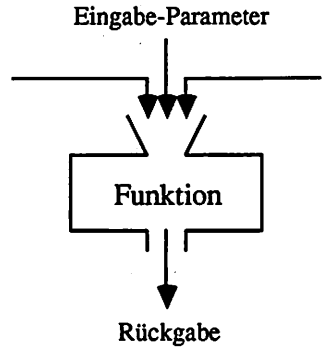
Variablen, die Sie ausdrücklich zur Änderungen durch Prozeduren freigeben wollen, sollten im Hauptprogramm als `PUBLIC` deklariert werden.

6 Anwender-definierte Funktionen (UDFs - user defined functions)

Eine der wohl wichtigsten Eigenschaften von Clipper ist die Möglichkeit, selbst Funktionen zu definieren. Dadurch kann der ohnehin große Sprachumfang von Clipper nach Belieben erweitert und den Wünschen und Erfordernissen des Programmierers angepaßt werden. UDFs können dabei in der Clipper-Sprache selbst, also aus "eingebauten" Anweisungen und Funktionen oder aus anderen selbst definierten Funktionen gebildet werden. Darüberhinaus ist es jedoch auch möglich, UDFs in C- oder in Assemblersprache zu schreiben und einzubinden. Dadurch werden die Möglichkeiten noch mehr erweitert. Man wird besonders dann darauf zurückgreifen, wenn zeitkritische Probleme zu lösen sind, oder solche, die in der Clipper Sprache nicht oder nur sehr aufwendig zu realisieren sind.

6.1 Was ist eine Funktion?

Eine Funktion kann man sich als eine "Verarbeitungsmaschine" vorstellen, also als einen "schwarzen Kasten", in den oben ggf. Werte eingegeben werden und der in jedem Fall unten einen bestimmten Wert zurückgibt. Die eingegebenen Werte bezeichnet man als *Eingabeparameter* während der zurückgegebene Wert (es ist stets *ein* Wert) als Rückgabe bezeichnet wird.



Während eine Funktion nicht unbedingt einen oder mehrere Eingabeparameter haben muß, wird stets genau ein Wert zurückgegeben. Normalerweise sind die innerhalb der Funktion ablaufenden Vorgänge und die darin verarbeiteten Werte für das übrige Programm "unsichtbar". In Clipper gibt es allerdings Ausnahmen von dieser Regel, die man jedoch tatsächlich nur in Ausnahmefällen ausnutzen sollte.

Die Schreibweise beim Aufruf einer Funktion entspricht folgendem Schema:

```
Variable = Funktionsname([Param.1, Param.2 … , Param.n])
```

Die Funktion wird von anderen Funktionen durch einen eindeutigen Namen unterschieden (`Funktionsname`). Hinter dem Namen werden der oder die Eingabeparameter angegeben. Die Parameter können Konstanten, Variablen oder auch Ausdrücke bzw. andere Funktionen sein. Sind mehrere Parmeter vorhanden, so müssen diese jeweils durch ein Komma getrennt werden. Die Zusammenfassung aller Parameter bezeichnet man auch als *Parameterliste*. Die Parameterliste muß in ein rundes Klammernpaar gesetzt werden, das *ohne* Leerzeichen auf den Funktionsnamen folgt. Ist kein Eingabeparameter erforderlich, so *muß* ein leeres Klammernpaar angegeben werden, da der Compiler sonst nicht erkennt, daß es sich um eine Funktion handelt.

> *Anmerkung*: Weglassen des leeren Klammernpaars ist ein "beliebter" Fehler, der auch bei "eingebauten" Funktionen gerne gemacht wird. Fatal ist, daß der Compiler (bedingt durch die dBASE/Clipper-Syntax) diesen Fehler nicht erkennen kann und stattdessen annimmt, daß es sich um einen Variablennamen handelt. So wird z.B. die Programmzeile
>
> ```
> IF .NOT. EOF
> ```
>
> ohne Fehlermeldung übersetzt und der Compiler nimmt an, daß die IF-Bedingung von der "Variablen" EOF gesteuert wird. Während der Programmausführung erhält man dann einen Laufzeitfehler (undefined EOF). Richtig muß die Zeile lauten:
>
> ```
> IF .NOT. EOF()
> ```

Da eine Funktion stets einen Wert zurückgibt, sollte diese normalerweise in einem Ausdruck stehen. In der obigen Darstellung steht die Funktion in einem Zuweisungs-Ausdruck, d.h. `variable` erhält einen Wert zugewiesen, der der Rückgabe der Funktion entspricht. Die Funktion kann natürlich auch in anderer Weise in einem Ausdruck verwendet werden, z.B.:

```
@ 5,10 SAY 2 * TESTFUNKT(6,3)
```

Ab der Clipper-Version Herbst ´86 ist es außerdem erlaubt, die Rückgabe einer Funktion "ins Leere" gehen zu lassen, d.h. eine Funktion kann alleine in einer Programmzeile stehen und muß nicht in einem Ausdruck enthalten sein. So bewirkt z.B. die nachstehende Programmzeile eine Verzögerung von 5 Sekunden, falls zuvor keine Taste gedrückt wird:

```
INKEY(5)
```

Natürlich gibt die (eingebaute) `INKEY()`-Funktion einen Wert zurück, nämlich entweder den Code einer gedrückten Taste oder den Wert 0, falls kein Tastendruck erfolgte. Dieser Wert ist aber im Programm nicht von Bedeutung, wenn es nur auf die Verzögerung ankommt.

Wenn im folgenden Text der Name einer Funktion erwähnt wird, wird er stets mit einem nachgestellten, leeren Klammernpaar angegeben, damit sofort ersichtlich ist, daß eine Funktion gemeint ist, z.B. `TEST1()`.

6.2 Wie wird eine Funktion definiert?

Die Definition einer Funktion beginnt stets mit der Zeile

```
FUNCTION Funktionsname
```

Danach folgt ggf. die Aufzählung der Eingabeparameter in der Zeile

PARAMETERS $Name_1$, $Name_2$ ... , $Name_n$

Wenn keine Parameter an die Funktion übergeben werden müssen, dann kann diese Zeile natürlich entfallen.

Im Anschluß daran folgen Programmzeilen, die definieren, was die Funktion ausführen soll und schließlich wird die Funktion mit der Zeile

```
RETURN Wert
```

beendet. Die `RETURN`-Anweisung *muß* in jedem Fall zusammen mit einem *Rückgabewert* angegeben werden. Hierin unterscheidet sich eine Funktionsdefinition von der - ansonsten gleichartigen - Prozedurdefinition.

Hier nochmals eine Zusammenfassung:

FUNCTION Funktionsname
PARAMETERS $Name_1$, $Name_2$ ... , $Name_n$

```
*
* Anweisungen
*
RETURN Wert
```

Es ist durchaus zulässig, daß eine Funktion mehrere RETURN-Anweisungen enthält, wie folgendes Beispiel zeigt:

```
*
* den größeren von zwei Werten zurückgeben
*
FUNCTION MAXVAL
PARAMETERS z1, z2
IF z1 > z2
  RETURN z1
ELSE
  RETURN z2
ENDIF
```

Wie Sie sehen, muß die RETURN-Anweisung nicht unbedingt die letzte Anweisung in einer Funktion sein. Übersichtlicher ist es jedoch, wenn eine Funktion mit einer einzigen RETURN-Anweisung beendet wird, die am Schluß der Funktionsdefinition steht. Die obige Funktion könnte ebensogut auch wie folgt definiert werden:

```
*
* den größeren von zwei Werten zurückgeben
*
FUNCTION MAXVAL
PARAMETERS z1, z2
RETURN IF(z1>z2,z1,z2)
```

6.3 Private Variablen in Funktionen

Sehr oft ist es erforderlich, innerhalb einer Funktionsdefinition Zwischenergebnisse in Variablen zu speichern. Dabei ist es wichtig, auf den Gültigkeitsbereich von Variablen zu achten. Wie auch bei Prozeduren gilt, daß eine Variable, die in einer höheren Programm-Ebene definiert wurde auch in den untergeordneten Ebenen definiert ist und dort geändert werden kann. Diese Eigenschaft kann in bestimmten Fällen nützlich sein, jedoch birgt sie gleichzeitig auch Fehlermöglichkeiten. Hierzu ein Beispiel:

```
PUBLIC e
a = 5
b = 2
c = 10
d = 20
f = 44
? "FKT(a,b,@f): "
?? FKT(a,b,@f)
? "a:        "
?? a
? "b:        "
?? b
? "c:        "
?? c
? "d:        "
?? d
? "e:        "
?? e
? "f         "
?? f
? "g         "
?? g
QUIT
FUNCTION FKT
PARAMETERS a, x, y
PRIVATE c
a = a + x
c = 2
d = 99
e = 777
y = 555
f = 999
RETURN a
```

Das Programm erzeugt folgende Ausgabe:

```
FKT(a):    7
a:         5
b:         2
c:        10
d:        99
e:       777
f:       555
g:
```

hier bricht das Programm mit der Fehlermeldung `undefined identifier g` ab.

Aus diesem Ergebnis kann man einige wichtige Schlußfolgerungen ziehen:

- Es ist unwichtig, welche Namen man in der PARAMETERS-Zeile der Funktion angibt. Diese Namen müssen nicht mit den Namen identisch sein, die beim Funktionsaufruf angegeben werden (in diesem Beispiel wurde zwar in beiden Fällen der Name `a` verwendet, aber als zweiter Parameter hat in der Funktion den Namen `x`, beim Aufruf jedoch den Namen b. Ebenso lautet der dritte Parameter `f`, innerhalb der Funktion jedoch `y`.

  Die Namen in der PARAMETERS-Zeile der Funktion dienen gewissermaßen als Platzhalter, die die Werte der übergebenen Parameter aufnehmen und der Funktion zur Verarbeitung übergeben.

- Werden in der Funktion Werte von übergebenen Parametern verändert, so wirkt sich das nicht auf die Inhalte der beim Aufruf als Parameter angegebenen Variablen aus. Im obigen Beispiel wird der Parameter `a` innerhalb der Funktion geändert (`a = a + x`), dennoch ist der Wert der Variablen `a` im Hauptprogramm nach Aufruf der Funktion unverändert, wie die Ausgabeanweisung `?? a` beweist. Das bedeutet, daß an eine Funktion standardmäßig die Übergabe als Wert und nicht (wie bei Prozeduren) per Referenz erfolgt.

- Wird in einer Funktion der Inhalt einer Variablen verändert, die im aufrufenden Programm bereits definiert war, so ist deren Inhalt nach Verlassen der Funktion auch im aufrufenden Programm geändert. Im Beispiel ist das bei der Variablen `d` der Fall. Der Variablen wurde zunächst der Wert 20 zugewiesen. In der Funktion erhält sie den neuen Wert 99, der auch nach Verlassen der Funktion gilt, wie die Programmzeile `?? d` beweist.

- Wird in einer Funktion eine Variable als PRIVATE ("privat") deklariert, so gilt sie nur innerhalb der Funktion. Variablen gleichen Namens im aufrufenden Programm werden nicht verändert, wenn Änderungen in der Funktion erfolgen. Das können Sie im Beispiel an der Variablen c verfolgen. Im Hauptprogramm wird ihr der Wert 10 zugewiesen. Obwohl in der Funktion eine gleichnamige, aber als PRIVATE deklarierte Variable c verwendet wird und den Wert 2 zugewiesen bekommt, bleibt der ursprüngliche Wert von c erhalten, wie die Anweisung ?? c zeigt.

- Eine im Hauptprogramm als PUBLIC dekarierte Variable ist auch in der Funktion gültig und kann durch die Funktion verändert werden (im Beispiel ist das die Variable e.

- Soll an eine Funktion ein Wert per Referenz, d.h. als Adresse übergeben werden, muß dem Namen in der Parameterliste beim Funktionsaufruf das Zeichen @ vorangestellt werden. Dann kann auch eine Funktion den ursprünglichen Inhalt der Variablen ändern, wie am Beispiel der Variablen f ersichtlich wird, die an die Funktion per Referenz an den "Platzhalter" y übergeben wird.

- Eine in der Funktion neu definierte Variable hat nur innerhalb der Funktion Gültigkeit, nicht jedoch im aufrufenden Programm. Obwohl in der obigen Funktion FKT() der Variablen g der Wert 999 zugewiesen wurde, ist sie danach im Hauptprogramm undefiniert, wie man an der Laufzeit-Fehlermeldung erkennt.

Nach der Definition sollte eine Funktion ein "schwarzer Kasten" sein, der bei Eingabe bestimmter Werte einen entsprechenden Wert zurückgibt; dabei muß das "Innere" des Kastens für das übrige Programm "unsichtbar" sein und darf dieses auch nicht beeinflussen. Damit Funktionen universell eingesetzt werden können, müssen alle intern verwendeten Variablen *unbedingt* als PRIVATE deklariert werden, um Konflikte mit gleichnamigen Variablen in aufrufenden Programmteilen zu vermeiden. Nur in Ausnahmefällen sollten Funktionen dazu verwendet werden, Variableninhalte im aufrufenden Programm zu ändern.

Im Gegensatz zu Prozeduren werden Parameter an Funktionen im Normalfall immer als Wert und nicht per Refernz übergeben, d.h. wird innerhalb der Funkton der Inhalt einer Parameter-Variablen verändert, wirkt sich das nicht auf die Werte im aufrufenden Programm aus. Soll eine Variable explizit per Referenz übergeben werden, muß ihrem Namen in der Parameterliste beim Aufruf der Funktion das Zeichen @ vorangestellt werden.

7 Anwender-definierte Funktionen in Clipper-Sprache

In diesem Kapitel werden nützliche Funktionen gezeigt, die lediglich Clipper-Anweisungen und Funktionen sowie andere, hier beschriebene Funktionen verwenden. Sie finden diese Funktionen als Quellprogramme und in der Bibliothekdatei `TT87.LIB` auch auf der beigefügten Diskette.

7.1 Position einer Zeichenkette in einer anderen Zeichenkette abfragen, mit Angabe der Startposition

```
*
* Funktion gibt die Position einer Teil-Zeichenkette in einer
* anderen Zeichenkette zurück, entsprechend AT(); hier kann
* jedoch angegeben werden, ab welcher Position die Suche be-
* ginnen soll, während bei AT() immer vom Anfang aus gesucht
* wird. Ist die Teilzeichenkette nicht enthalten, wird der
* Wert 0 zurückgegeben.
*
* Parameter:  ch   =: gesuchte Teil-Zeichenkette
*             txt  =: abzusuchende Zeichenkette
*             s    =: Start-Position
*
FUNCTION ch_at
PARAMETERS ch, txt, s
  PRIVATE p
  p = AT(ch,SUBSTR(txt,s))
RETURN if(p=0,0,p+s-1)
```

Diese Funktion ist nützlich, wenn festgestellt werden soll, ob ab einer bestimmten Position in einer Zeichenkette eine andere Teil-Zeichenkette enthalten ist. In der weiter unten beschriebenen UDF `ASK()` sehen Sie ein Beispiel für die Verwendung von `ch_at()`.

7.2 Tastendruck lesen und solange den Cursor ausschalten

```
*
* Funktion arbeitet wie INKEY(); schaltet jedoch den Cusor
* während der Wartezeit aus
*
* Parameter:  zeit =: Wartezeit in Sekunden (0 =: unendlich)
*
* Rückgabe:   Code der gedrückten Taste
*
* Anmerkung:  Obwohl diese Clipper-Version die Anweisungen
*             SET CURSOR ON und SET CURSOR OFF "kennt", wird
*             hier die interne Prozedur __setctyp verwendet,
*             da diese korrekt den internen Cursor-Status
*             steuert.
*
FUNCTION waitkey
PARAMETERS zeit
  PRIVATE c, c_stat
  c_stat = IF(is_curs(),1,0)          && Cursor-Zustand abfr.
  CALL __setctyp WITH WORD(0)         && Cursor aus
  c = INKEY(zeit)                     && Taste lesen
  CALL __setctyp WITH WORD(c_stat)    && alter Cursorzustand
RETURN c
```

Diese Funktion kann anstelle von INKEY() verwendet werden. Sie hat den Vorteil, daß der oftmals störende Cursor während der Wartezeit abgeschaltet wird. Die aufgerufende Routine __SETCTYP ist als interne Funktion in der Clipper-Library enthalten. Sie wird hier verwendet, obwohl Clipper die Anweisungen SET CURSOR ON/OFF "kennt". Wie sich gezeigt hat, setzen die SET-Anweisungen nicht immer korrekt das interne Cusor-Statusflag, so daß die an anderer Stelle beschriebene Funktion IS_CURS() fehlerhafte Werte liefert.

Mit der Funktion IS_CURS() (siehe Kapitel 9 - Assembler-UDFs) wird abgefragt, ob der Cusor beim Aufruf der Funktion sichtbar ist und dieser Zustand gespeichert. Vor Verlassen der Funktion wird der alte Cursor-Zustannd wiederhergestellt.

7.3 Anzeigen einer Fehlermeldung mit Quittung

```
*
* Funktion gibt eine Fehlermeldung auf dem Bildschirm in einem
* automatisch zentrierten Rahmen aus und wartet auf einen
* Tastendruck.
*
* Der Bildschirm wird gesichert und nach dem Tastendruck in
* den alten Zustand versetzt
*
* Parameter:  zeile =: unterste Zeile für Anzeigefenster
*             mldg  =: anzuzeigender Text, maximal 76 Zeichen
*
* Rückgabe:   Code der gedrückten Taste
*
FUNCTION f_meldung
PARAMETERS zeile, mldg
  PRIVATE lg, li, re, s_puffer, c
  CLEAR TYPEAHEAD                    && Tastaturpuffer löschen
  lg = len(mldg)
  IF lg > 76                         && Länge der Meldung
*                                       prüfen
    lg = 76                          && und ggf. kürzen
    mldg = SUBSTR(mldg,1,76)
  ENDIF
  li = INT((80-lg)/2)-2              && linke Fenster-Spalte
  re = li + lg + 3                   && rechte Fenster-Spalte
*                                    && Fenster sichern
  s_puffer = SAVESCREEN(zeile-2,li,zeile,re)
  TONE(2000,3)                       && Pips !
  @ zeile-2,li,zeile,re BOX ;
    CHR(201)+CHR(205)+ ;
    CHR(187)+CHR(186)+ ;
    CHR(188)+CHR(205)+ ;
    CHR(200)+CHR(186)+" "            && Doppel-Rahmen zeichnen
  @ zeile-1,li+2 SAY mldg            && Text ausgeben
  c = WAITKEY(0)                     && Warten auf Taste
*                                    && Schirm zurückholen
  RESTSCREEN(zeile-2,li,zeile,re,s_puffer)
RETURN c                             && Tastencode zurückgeben
```

Diese Funktion läßt sich fast in allen Programmen verwenden, um eine Fehlermeldung für den Bediener anzuzeigen. Da die Funktion den Code der Taste zurückgibt, mit der der Bediener die Meldung quittiert hat, kann man ggf. je nach Tastendruck unterschiedlich reagieren.

Beispiel für den Funktionsaufruf:

```
...
SEEK name
IF !FOUND()
  f_meldung(20,"Kunde nicht gespeichert")
ENDIF
...
```

7.4 Text-Ausgabe mit Ja/Nein-Abfrage

```
* Funktion gibt Abfrage-Text in einem automatisch zentrierten
* Rahmen aus und erwartet "J/N"-Eingabe. Der Bildschirm wird
* automatisch gesichert
*
* Parameter: zeile =: unterste Zeile für Anzeigefenster
*            txt   =: Abfrage-Text, " (J/N) ?" wird
*                     automatisch angefügt, maximal 68 Zeichen
*
* Rückgabe:  .T.   =: "J" oder "j" gedrückt
*            .F.   =: "N" oder "n" gedrückt
*
FUNCTION jn_meldung
PARAMETERS zeile, mldg
  PRIVATE l, li, re, ch, s_puffer
  CLEAR TYPEAHEAD                    && Tast.-Puffer löschen
  lg = len(mldg)
  IF lg > 68                         && Länge prüfen
    lg = 68                          && und ggf. kürzen
    mldg = SUBSTR(mldg,1,68)
  ENDIF
  mldg = mldg + " (J/N) ?"           && Text ergänzen
  li = INT((80-lg)/2)-2              && linke Fenster-Spalte
  re = li + lg + 3                   && rechte Fenster-Spalte
*                                    && Schirm sichern
  s_puffer = SAVESCREEN(zeile-2,li,zeile,re)
```

```
  @ zeile-2,li,zeile,re BOX ;
    CHR(201)+CHR(205)+ ;
    CHR(187)+CHR(186)+ ;
    CHR(188)+CHR(205)+ ;
    CHR(200)+CHR(186)+" "              && Doppel-Rahmen zeichnen
  @ zeile-1,li+2 SAY mldg              && Text ausgeben
  ch = " "
  DO WHILE !(ch $ "JN")                && Warten, bis "J", "j",
    ch = UPPER(CHR(WAITKEY(0)))        && "N" oder "n" ist
  ENDDO
*                                      && Schirm zurückholen
  RESTSCREEN(zeile-2,li,zeile,re,s_puffer)
RETURN ch = "J"                        && .T., wenn ja
```

Diese Funktion kann immer dann verwendet werden, wenn der Bediener eines Programms eine bestimmte Abfrage mit "J" oder "N" bestätigen soll. Wichtig ist, daß nur die Tasteneingaben "J", "j", "N" und "n" angenommen werden. Bei allen anderen Eingaben wird die Funktion nicht verlassen.

Beispiel für den Funktionsaufruf:

```
-
IF jn_meldung(20,"Eingabe beenden")
  EXIT
ENDIF
...
```

Anmerkung: Es ist gefährlich, wenn man nur die positive Antwort ("J" oder "j") allein auswertet und bei allen anderen Tasteneingaben eine negative Antwort annimmt. Stellen Sie sich vor, daß nach Erfassung größerer Datenmengen die Frage

"Eingaben speichern (J/N) ?"

gestellt wird. Wenn nun der Bediener versehentlich "k" statt "j" tippt, gehen alle Eingaben verloren, wenn diese Eingabe als negative Antwort interpretiert würde.

7.5 Ausgabe eines Abfragetextes mit mehreren Antwortmöglichkeiten

```
*
* Funktion zeigt in einem zentrierten Rahmen einen Abfrage-
* Text an und wartet danach auf einen Tastendruck. Als
* gültige Eingabezeichen werden solche Zeichen angenommen,
* die im Abfragetext mit einer vorangestellten Markierung
* versehen sind. Bei allen anderen Tasteneingaben wartet die
* Funktion weiter.
*
* Der Bildschirm-Inhalt wird automatisch gesichert und beim
* Verlassen der Funktion wiederhergestellt.
*
* Parameter:  zeile =: unterste Zeile für Anzeigefenster
*             msg   =: Abfragetext
*             delim =: Markierungszeichen
*
* Rückgabe:   eingegebenes Zeichen
*
FUNCTION ask
  PARAMETERS zeile, msg, delim
  PRIVATE l, li, re, ch, msk, puffer
  CLEAR TYPEAHEAD                    && Tastaturpuffer löschen
  l = LEN(msg)                       && Länge des Abfagetextes
  li = INT((80-l)/2)-2               && linke Fensterpalte
  re = li + 3 + l                    && rechte Fensterspalte
*                                    && "alten" Schirm sichern
  puffer = SAVESCREEN(zeile-2,li,zeile,re)
  @ zeile-2,li,zeile,re BOX ;
    CHR(201)+CHR(205)+ ;
    CHR(187)+CHR(186)+ ;
    CHR(188)+CHR(205)+ ;
    CHR(200)+CHR(186)+" "            && Doppel-Rahmen zeichnen
  @ zeile-1,li+2 say msg             && Text ausgeben
  msk = ""                           && Zeichenkette für gül-
*                                       tige Eingabezeichen
*                                       initialisieren
  p = AT(delim,msg)                  && Position des ersten
*                                       Markierungszeichens
```

```
  IF p = 0                                  && Zeichen nicht vorh.
    RETURN("")                              && Leerstring zurückgeben
  ENDIF
  DO WHILE p <> 0                           && Markierungszeichen
*                                              suchen
    msk = msk + ;
          UPPER(SUBSTR(msg,p+1,1))          && Gültig. Eingabezeichen
*                                              anfügen
    p = ch_at(delim,msg,p+2)                && noch ein Markierungs-
*                                              zeichen ?
  ENDDO
  ch = " "                                  && auf gült. Eingabezei-
  DO WHILE !(ch $ msk)                      && chen warten
    ch = UPPER(CHR(waitkey(0)))
  ENDDO
*                                           && "alten" Schirm zurück
  RESTSCREEN(zeile-2,li,zeile,re,puffer)
RETURN ch
```

Diese Funktion vereinfacht Programme, in denen Abfragen enthalten sind, wie z.B.:

```
Konto <N>eu anlegen, <A>endern, <L>öschen oder <E>nde
```

Setzen Sie in Ihr Progamm den Aufruf

```
wahl = ask(20," Konto <N>eu anlegen, <A>endern, " + ;
                "<L>öschen oder <E>nde","<")
```

und die Funktion `ask()` erledigt das Anzeigen des Textes, die Prüfung der erlaubten Eingaben und die Umwandlung in Großbuchstaben.

Es muß beim Aufruf der Funktion natürlich sichergestellt werden, daß alle gewünschten Eingabezeichen im Abfragetext durch das gleiche Markierungszeichen gekennzeichnet werden. Ist das angegebene Markierungszeichen nicht im Abfragetext enthalten, gibt die Funktion eine leere Zeichenkette zurück.

7.6 Datum auf Gültigkeit prüfen

```
*
* Prüfung, ob Zeichenkette ein gültiges Datum enthält und ggf.
* Fehlermeldung ausgeben
*
* Parameter:  ds  =: Datum-String der Form "tt.mm.jj"
*
* Rückgabe:   .T. =: Datum ist gültig
*             .F. =: Datum ist ungültig (nach vorangehender
*                    Fehlermeldung)
*
* WICHIG:     vorher SET DATE GERMAN ausführen !

FUNCTION datum_ok
PARAMETERS ds
  IF SUBSTR(DTOC(CTOD(ds)),1,1) = " "
    f_meldung(20,"Ungültiges Datum")
    RETURN .F.
  ELSE
    RETURN .T.
  ENDIF
```

Diese Funktion ist nützlich, um ein Datum auf Gültigkeit zu prüfen. Clipper besitzt zwar auch eine interne Gültigkeitsprüfung von Datum-Variablen, diese führt aber zur festen Fehlermeldung "`Invalid date`" in Zeile 0. Bei dieser Funktion können Sie jedoch an beliebiger Stelle einen freien Text anzeigen. Beachten Sie, daß hier keine Datum-Variable, sondern eine Text-Eingabe erwartet wird.

Die Prüfung des Datums ist etwas "trickreich". Bei ungültigen Datum-Angaben (z.B. 32.06.87) liefert die Funktion `CTOD()` ein leeres Datum, das mit `DTOC()` anschließend wieder in eine Zeichenkette umgewandelt wird. Hier ist es dann nur noch nötig, die erste Stelle dieser Zeichenkette auf ein Leerzeichen zu prüfen, um zu erkennen, ob das Datum ungültig ist.

Beispiel für den Funktionsaufruf:

```
-
dt = "  .  .  "
@ 10, 0 SAY "Tagesdatum:"
DO WHILE .T.
  @ 10,12 GET dt PICTURE "99.99.99"
  READ
  IF datum_ok(dt)
    EXIT
  ENDIF
ENDDO
...
```

"Elegante" Lösung mit VALID-Klausel:

```
-
dt = "  .  .  "
@ 10, 0 SAY "Tagesdatum:"
@ 10,12 GET dt VALID datum_ok(dt)
READ
...
```

7.7 Zeit auf Gültigkeit prüfen

```
*
* Prüfung, ob String eine gültige Zeitangabe enthält und ggf.
* Fehlermeldung ausgeben
*
* Parameter:  ts  =: Zeit-String der Form "hh:mm"
*
* Rückgabe:   .T. =: Zeit ist gültig
*             .F. =: Zeit ist ungültig (nach vorangehender
*                    Fehlermeldung)
*
FUNCTION zeit_ok
PARAMETERS ts
  PRIVATE ok, t
  ok = .T.                          && gültige Zeit annehmen
  t = SUBSTR(ts,1,2)
  IF (t < "00") .OR. (t > "23")     && Stunden prüfen
    ok = .F.
  ENDIF
```

```
  t = SUBSTR(ts,4,2)
  IF (t < "00") .OR. (t > "59")        && Minuten prüfen
    ok = .F.
  ENDIF
  IF !ok                               && Ungültige Zeit
    f_meldung(20,"Ungültige Zeit")     && melden
  ENDIF
RETURN ok
```

Ähnlich wie die vorangehende Funktion zur Prüfung eines Datums wird hier eine Zeitangabe geprüft. Bei Bedarf können Sie die Funktion auch auf die Prüfung der Sekundenstellen erweitern.

Beispiel für den Funktionsaufruf:

```
-
zt = "  :  "
@ 10, 0 SAY "Zeit:"
DO WHILE .T.
  @ 10,12 GET zt PICTURE "  :  "
  READ
  IF zeit_ok(zt)
    EXIT
  ENDIF
ENDDO
...
```

7.8 Prüfung auf "leere" String-Variable

```
* Prüfen, ob ein String-Wert "leer" ist und ggf. Fehlermeldung
* ausgeben
*
* Parameter:  txt   =: String-Inhalt
*             msk   =: Maske für leeres Feld
*
* Rückgabe:   .T.   =: String ist "leer"
*             .F.   =: String ist nicht "leer"
*
FUNCTION leer_str
PARAMETERS txt, msk
  PRIVATE ret
  IF EMPTY(msk)                        && Maske leer, nur
    ret = !EMPTY(txt)                  && prüfen, ob Leerstring
  ELSE                                 && sonst prüfen, ob
    ret = (txt != mask)                && String = Maske
  ENDIF
  IF !ret
    f_meldung(20,"Feld darf nicht leer bleiben")
  ENDIF
RETURN ret
```

Diese Funktion ist besonders dann nützlich, wenn in einer Eingabemaske bestimmte Felder nicht "leer" sein dürfen. "Leer" bedeutet dabei nicht unbedingt daß das Feld nur Leerzeichen enthält, sondern u.U. auch, daß ein bestimmtes Grundraster (z.B. " / / - ") nicht mit Zeichen gefüllt ist. Die Funktion erwartet daher als zweiten Parameter die Angabe eines Masken-String, der ein solches Raster enthalten kann. Wird keine Maske benötigt, kann hier ein Leer-String angegeben werden.

Beispiel für den Funktionsaufruf:

```
...
name = SPACE(20)
STORE "  /  /  -  " TO kdnr, maske
@ 10, 0 SAY "Name:"
@ 12, 0 SAY "Kd-Nr:"
@ 10,15 GET name VAILD leer_str(name,"")
@ 12,15 GET kdnr PICTURE maske ;
        VALID leer_str(kdnr,maske)
READ
```

7.9 Bereichsprüfung eines numerischen Wertes

```
* Prüfen, ob ein numerischer Wert in einem best. Bereich liegt
* und ggf. Fehlermeldung ausgeben
*
* Paramter: num  =: zu prüfender Wert
*           low  =: unterer Grenzwert
*           high =: oberer Grenzwert
*
* Rückgabe: .T.  =: Wert ist im Bereich
*           .F.  =: Wert liegt außerhalb
*
FUNCTION in_range
PARAMETERS num, low, high
  PRIVATE ret
  ret = (num >= low) .and. (num <= high)
  IF !ret
    f_meldung(20, "Nur Zahlen von " + ;
    LTRIM(STR(low)) + " bis " + ;
    LTRIM(STR(high)) + " eingeben")
  ENDIF
RETURN ret
```

Diese Funktion arbeitet ähnlich wie die RANGE-Klausel bei einer GET-Anweisung, sie hat aber den Vorteil, daß eine Fehlermeldung an beliebiger Bildschirm-Position ausgegeben werden kann.

Beispiel für den Funktionsaufruf:

```
...
n = 0
@ 10, 0 SAY "Stückzahl"
@ 10,15 GET n PICTURE "9999" VALID in_range(n,0,1200)
READ
...
```

7.10 Zeitdifferenz berechnen

```
* Funktion berechnet Differenz zwischen zwei Zeiten,
* ein Tageswechsel wird berücksichtigt (maximal 24 Stunden)
*
* Parameter: t1 =: Anfangszeit, String im Format "hh:mm"
*            t2 =: Endzeit, String im Format "hh:mm"
*
* Rückgabe:  Zeitdifferenz in Minuten als numerischer Wert
*
FUNCTION t_diff
PARAMETERS t1, t2
  PRIVATE m1, m2
  m1 = VAL(SUBSTR(t1,1,2)) * 60 + ;
       VAL(SUBSTR(t1,4,2))            && Anfangszeit in Minuten
  m2 = VAL(SUBSTR(t2,1,2)) * 60 + ;
       VAL(SUBSTR(t2,4,2))            && Endzeit in Minuten
  IF m2 < m1                          && Wenn Tageswechsel,
    m2 = m2 + 1440                    && Endzeit korrigieren
  ENDIF
RETURN(m2-m1)                         && Differenz zurückgeben
```

Diese Funktion kann allgemein verwendet werden, wenn Zeitdifferenzen zu berechnen sind. Sie kann auf Wunsch auf die Sekundenstelle erweitert werden. Wichtig ist, daß auch die Zeitdifferenz bei einem Tageswechsel (z.B. Anfangszeit = 22:54, Endzeit = 02:32) berücksichtigt wird.

Beispiel für den Funktionsaufruf:

```
...
an = "02:34"
ab = "21:45"
? t_diff(ab, an)
...
```

7.11 Potenzberechnung

```
* Funktion berechnet n^x
*
* Parameter: n := Mantisse
*            x := Exponent
*
* Rückgabe:  n hoch x
*
FUNCTION pot
PARAMETERS n, x
IF n < 0
  RETURN(-EXP(LOG(ABS(n)) * x))
ELSE
  RETURN(EXP(LOG(n) * x))
```

7.12 Zentrierte Ausgabe eines Textes

```
* Funktion gibt einen Text zentriert innerhalb einer Bild-
* schirm-Zeile aus.
*
* Parameter: zeile =: Ausgabezeile (0…24)
*            txt   =: auszugebender Text
*
FUNCTION center
PARAMETERS zeile,txt
  @ zeile,int((80-len(txt))/2) say txt
RETURN .t.
```

Diese einfache Funktion erleichtert die Ausgabe von Bildschirm-Titeln. Eine Anwendung finden Sie in Kapitel 17 (Programmbeispiele) u.a. im Programm "BROWSE".

7.13 Rekursive Funktionen

Clipper-UDFs können auch rekursiv geschrieben werden. Das bedeutet, daß eine UDF sich selbst aufrufen kann. Durch rekursive Funktionen kann man oft mit verblüffend wenig Anweisungen umfangreiche Operationen ausführen.

Beim Entwurf einer solchen Funktion sollte man nach folgendem Schema vorgehen:

1. Festlegen einer Abbruchbedingung - irgendwann muß der Zyklus des Selbst-Aufrufes unterbrochen werden, damit die Funktion nicht endlos läuft.

2. Festlegen eines Rekursions-Schrittes; das ist meistens der Schluß von einem Fall auf den nächst einfacheren.

Hier ein Beispiel:

Es sollen die Fakultäten von Zahlen berechnet werden. Dazu gilt z.B.:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
4! =     4 * 3 * 2 * 1 =  24
1! =                 1 =   1
```

Wie Sie aus der Berechnung von `5!` und `4!` erkennen können, läßt sich `5!` auch wie folgt ausdrücken:

```
5! = 5 * (4 * 3 * 2 * 1) = 120 oder
5! = 5 *  4!             = 120
```

Damit hat man den Schluß von einem Fall auf den nächst einfacheren, den man allgemein so ausdrücken kann:

```
n! = n * (n-1)!
```

Nun fehlt noch die Abbruch-Bedingung, die jedoch bereits oben steht:

```
1! = 1
```

Daraus kann man sofort eine entsprechende UDF definieren:

```
FUNCTION FAKT
PARAMETERS n
IF n <= 1                          && Abbruch-Bedingung
  RETURN(1)
ELSE
  RETURN(n * fakt(n-1))            && Rekursions-Schritt
ENDIF
```

Hier wurde die Abbruchprüfung nicht mit der Anweisung `IF n = 1`, sondern mit `IF n <= 1` gebildet, um den Sonderfall `0! = 1`, der per Definition gilt, ebenfalls zu berücksichtigen.

Versuchen Sie einmal, eine Funktion zu schreiben, die ebenfalls Fakultäten berechnet, die jedoch ohne Rekursionen arbeitet. Sie werden bestimmt mehr Anweisungen benötigen.

8 Anwender-definierte Routinen in anderen Sprachen

Ein großer Vorteil von Clipper ist es, daß Routinen in Clipper-compilierte Programme eingebunden werden können, die nicht in Clipper-Sprache, sondern in C- oder Assemblersprache geschrieben sind. In der Clipper-Library sind Funktionen enthalten, die das Einbinden solcher Routinen vereinfachen und die eine standardisierte Übergabe von Parametern und die Rückgabe von Werten ermöglichen. Diese "Programmschnittstellen" werden auch bei zukünftigen Clipper Versionen beibehalten, so daß ein Wechsel auf eine neue Version ohne großen Aufwand möglich ist.

Obwohl es auch andere Wege gibt, Werte an eine C- oder Assembler-Routine zu übergeben bzw. von ihr zurückzugeben, wird dringend empfohlen, *nur* die Programmschnittstellen zu verwenden, um keine Programmfehler zu erhalten und um kompatibel zu späteren Clipper Versionen zu bleiben.

In diesem Abschnitt werden die Übergabe- und Rückgaberoutinen allgemein beschrieben. In den nachfolgenden Abschnitten finden Sie nähere Angaben über deren Verwendung in C bzw. Assembler.

Beachten Sie, daß alle Namen dieser Routinen mit einem doppelten Unterstrich beginnen. In Assembler-Routinen müssen die Namen exakt so angegeben werden. In C-Routinen werden die Namen nur mit einem vorangestellten Unterstrich angegeben, da Microsoft-C automatisch bei der Bildung der Objekt-Module selbständig einen weiteren Unterstrich anfügt.

8.1 Aufrufe zur Parameter-Übergabe

Für die verschiedenen Datentypen gibt es unterschiedliche Aufrufe, wie folgende Tabelle zeigt:

Fkt.-Name	Clipper-Datentyp	Übergabe an Routine
`__PARC()`	String	"Zeiger" auf String
`__PARDS()`	Datum	"Zeiger" auf Datum-String [1]
`__PARL()`	logisch	16-Bit-Wert (0 := .F., ungl. 0 := .T.)
`__PARNI()`	numerisch	16-Bit Ganzzahl[2]
`__PARNL()`	numerisch	32-Bit Gleitkommazahl (long)[3]
`__PARND()`	numerisch	64-Bit Gleitkommazahl (double)[3]

Alle Routinen zur Parameter-Übernahme werden mit zwei Werten aufgerufen, wobei der zweite Wert optional ist und nur bei der Übernahme aus einem Array-Element benutzt wird.

Der erste Wert gibt die Position des gewünschten Wertes innerhalb der Parameter-Liste der UDF an. Wenn es sich dabei um ein Array handelt, muß mit dem zweiten Wert der Index des gewünschten Array-Elementes bestimmt werden.

Beispiele für die Verwendung der Routinen `__PARx()` finden Sie in den nachfolgenden Kapiteln.

8.2 Aufrufe zur Parameter-Rückgabe

Entsprechend zu Clipper-internen Aufrufen zur Parameter-Übergabe an Routinen gibt es auch Aufrufe zur Rückgabe von Parametern, die in folgender Tabelle zusammengestellt sind:

Fkt.-Name	Clipper-Datentyp	Rückgabe von Routine
`__RETC()`	String	"Zeiger" auf String, Ende = Null-Byte
`__RETCLEN()`	String	"Zeiger" auf String und Länge des Strings
`__RETDS()`	Datum	"Zeiger" auf Datum-String [1]
`__RETL()`	logisch	16-Bit-Wert (0 := .F., ungl. 0 := .T.)
`__RETNI()`	numerisch	16-Bit Ganzzahl[2]
`__RETNL()`	numerisch	32-Bit Gleitkommazahl (long)[3]
`__RETND()`	numerisch	64-Bit Gleitkommazahl (double)[3]
`__RET()`	ohne Wert	Rücksprung ohne Wert (void)

1) Die Clipper-interne Darstellung des Datums wird in eine Zeichenkette umgewandelt und der "Zeiger" auf diese Zeichenkette wird übergeben.

2) Intern stellt Clipper numerische Variablen immer als Gleitkommazahl dar. Da (besonders in Assemblersprache) diese Darstellung schwierig zu verarbeiten ist, besitzt Clipper die Übernahmefunktion __PARNI(). Diese wandelt einen numerischen Wert in eine 16-Bit Ganzzahl um, die dann leicht verarbeitet werden kann. Man muß jedoch darauf achten, daß die übergebenen Werte nicht den Bereich überschreiten, der sich in einem Wort darstellen läßt (0...65535 bzw. -32768...32767).

3) Diese numerischen Formate entsprechen den numerischen Formaten long und double von *Microsoft-C*. Sie sind also besonders bei C-Routinen einsetzbar.

8.3 Aufrufe zur Parameter-Prüfung

Zusätzlich zu den oben angegebenen Aufrufen zur Übernahme und Rückgabe von Parametern sind weitere Aufrufe in Clipper definiert, mit denen es möglich ist, die Anzahl der an die Routine übergebenen Parameter abzufragen. Außerdem kann damit auch der Typ jedes einzelnen Parameters, sowie die Länge von Zeichenketten abgefragt werden.

8.3.1 Funktion _PARINFO()

Diese Funktion hat die Bezeichnung `__PARINFO()`. Wird sie mit dem Parameter 0 aufgerufen, so gibt sie die Anzahl der beim Aufruf der UDF übergebenen Parameter zurück.

Soll der Datentyp eines an die UDF übergebenen Parameters abgefragt werden, muß `__PARINFO()` mit der Positionsnummer aufgerufen werden. So gibt z.B. `__PARINFO(1)` den Typ des ersten Parameters und `__PARINFO(3)` den Typ des dritten Parameters zurück. Die Parametertypen werden dabei durch einen 16-Bit Ganzzahlwert gemäß folgender Tabelle gekennzeichnet:

Wert dez.	Typ
0	nicht definiert
1	String
2	numerisch
4	logisch
8	Datum
16	Alias
32	Zeiger (mit Typ OR-verknüpft, wenn per Referenz übergeben)
65	Memo
128	16-Bit Ganzzahl
512	Array

Durch die Funktion `__PARINFO()` ist es möglich, daß UDFs überprüfen können, ob die richtige Anzahl von Parametern und die richtigen Typen übergeben wurden. Damit kann man verhindern, daß bei Aufruf mit fehlerhaften Parametern unsinnige Werte zurückgegeben werden oder gar das Programm "abstürzt". Es ist damit auch möglich, UDFs zu schreiben, die mit variabler Anzahl von Parametern aufgerufen werden können. Wenn z.B. eine Funktion

die Angabe eines Laufwerkskenners erwartet, so kann sie das aktuelle Laufwerk annehmen, wenn kein spezielles Laufwerk angegeben wurde. Hier ein Beispiel mit einer fiktiven Funktion `DISKTEST()`:

`DISKTEST("A:")`	spricht Laufwerk A: an
`DISKTEST()`	spricht das aktuelle Laufwerk an

8.3.2 Funktion _PARINFA()

Diese Funktion erlaubt es, ähnlich wie `__PARINFO()` bei Einzelparametern, Typ-Prüfungen an einzelnen Array-Elementen vorzunehmen.

Der Aufruf von `__PARINFA()` erfolgt mit zwei Parametern. Als erster Parameter wird, wie bei `__PARINFO()`, die Position in der Parameterliste angegeben; der zweite Parameter dient als Array-Index. Die Funktion gibt einen Ganzzahlwert zurück, der den Typ des jeweiligen Array-Elementes kennzeichnet. Die Bedeutung dieses Kenners ist identisch mit der bei `__PARINFO()`. Eine Typprüfung der einzelnen Array-Elemente muß möglich sein, da Clipper es zuläßt; daß die einzelnen Elemente eines Arrays Werte unterschiedlicher Typen enthalten können.

Gibt man als zweiten Parameter der Funktion `__PARINFA()` den Wert 0 an, liefert die Funktion die Anzahl der Elemente im Array zurück.

8.3.3 Funktion _PARCLEN()

Diese Funktion liefert die Länge einer Zeichenkette, in der auch Null-Zeichen (`CHR(0)`) enthalten sein dürfen. Dabei wird das abschließende Null-Zeichen nicht mitgezählt.

Auch diese Funktion muß mit einem Parameter aufgerufen werden, der die Position des gewünschten Elementes innerhalb der Parameter-Liste der UDF bestimmt. Wenn es sich dabei um ein Array handelt, muß als zweiter Parameter der Array-Index angegeben werden.

8.3.4 Funktion _PARCSIZ()

Diese Funktion gibt den für eine Zeichenkette belegten Speicherplatzes zurück. Dabei zählt das abschließende Null-Byte mit. Die Funktion ist dann wichtig, wenn die abgefragte Zeichenke geändert werden soll. Dabei ist es nämlich unbedingt wichtig, daß nicht über das Ende der Original-Zeichenkette hinaus geschrieben wird, da sonst andere Variablen-Inhalte zerstört werden.

`__PARCSIZ()` muß wie `__PARCLEN()` mit einem bzw. zwei Parametern aufgerufen werden.

8.4 Funktionen zur Speicher-Verwaltung

Häufig benötigen Assembler- oder C-Routinen Speicherbereiche, deren Größe erst während der Ausführung bestimmt werden kann. Aus diesem Grund bietet Clipper eine Routine, mit der dynamisch Speicherplatz angefordert werden kann. Mit einer anderen Routine kann nicht mehr benötigter Speicherplatz freigegeben werden.

8.4.1 Anforderung von Speicher: __EXMGRAB()

Diese Routine wird mit der Anzahl der benötigten Bytes aufgerufen und sie gibt einen Zeiger auf diesen Speicherbereich zurück, wenn er zur Verfügung gestellt werden kann. Steht im System nicht mehr genügend freier Speicher bereit, um die Anforderung zu erfüllen, gibt `__EXMGRAB()` den Wert 0 zurück.

8.4.2 Freigabe von Speicher: __EXMBACK()

Mit dieser Routine kann zuvor von `__EXMGRAB()` angeforderter Speicher wieder freigegeben werden. Die Funktion wird mit zwei Parametern aufgerufen. Zuerst gibt man den Zeiger auf den freizugebenden Bereich und dann die Anzahl der freizugebenden Bytes an. Die Funktion hat keine Rückgabe, d.h. sie ist vom Typ `void`.

9 Anwender-definierte Funktionen in Assemblersprache

Der Einsatz von Assembler-Routinen in Clipper-Programmen ist besonders dann interessant, wenn maschinennahe Funktionen ausgeführt werden sollen oder wenn eine hohe Ausführungsgeschwindigkeit verlangt wird.

9.1 Allgemeiner Aufbau einer UDF in Assembler-Sprache

Die nachfogenden Quellprogramme wurden für den Microsoft 8088/8086-Assembler MASM für MS-DOS, Ver. 4.0 geschrieben. Mit dieser Version wurden auch die Clipper-internen Assemblermodule entwickelt, so daß bei Verwendung dieses Assemblers keine Anpassungsprobleme auftreten werden. Allgemein sind auch ältere Versionen des Microsoft-Assemblers verwendbar, wie auch Produkte anderer Hersteller, sofern sie einen relozierbaren Objekt-Code erzeugen, der dem Microsoft-Standard entspricht.

Der Aufbau eines Quellprogramms für eine Assembler-UDF sollte nach folgendem Schema erfolgen:

1. PUBLIC-Deklaration für den Funktionsnamen
2. EXTERN-Deklarationen für externe Namen; hierzu gehören besonders die Namen der Clipper-internen Routinen zur Übernahme, Rückgabe und Prüfung von Parametern. Es empfiehlt sich, diese Standard-Deklarationen in eine INCLUDE-Datei zu setzen, wie weiter unten gezeigt.
3. Definition des Datensegments, falls erforderlich
4. Daten-Definition
5. Datensegment-Ende
6. Definition des Codesegments und der Segmentregister
7. Deklaration der Funktion als FAR-Prozess
8. Programmcode
   8.1 Sichern der Register
   8.2 Prüfen und Lesen von Übergabeparamtern (falls erforderlich)
   8.3 Anwendungs-abhängiger Code
   8.4 Alten Register-Inhalt wiederherstellen
   8.5 Rückgabewert auf den Stack bringen (falls Rückgabe nötig)
   8.6 Rückgabe- oder Rücksprungfunktion aufrufen
   8.7 Stack bereinigen, falls Wertrückgabe
   8.8 RET-Anweisung
9. Codesegment-Ende
10. Quellcode-Ende

Hier der Code für ein allgemeines Rahmenprogramm für Assembler-UDFs:

```
            PUBLIC   <UDF-Name>

            EXTRN    __PARC:FAR          ; hier alle benötigten
                                         ; Übernahme-Routinen
                                         ; angeben
            EXTRN    __RETC:FAR          ; hier alle benötigten
                                         ; Rückgabe-Routinen
                                         ; angeben
            EXTRN    __PARINFO           ; falls benötigt

DGROUP      GROUP    <DS-Name>           ; Datensegment dekla-
<DS-Name>   SEGEMENT PUBLIC '_DATA'      ; rieren, falls benötigt
;
;           Daten-Definitionen
;
<DS-Name>   ENDS

<CS-Name>   SEGMENT  'CODE'              ; Codesegment
                                         ; deklarieren
            ASSUME   CS:<CS-Name>        ; Segmentregister
            ASSUME   DS:<DS-Name>        ; definieren

<UDF-Name>  PROC     FAR                 ; Funktion ist
                                         ; FAR-Prozess

            PUSH     BP                  ; Register sichern
            MOV      BP,SP
            PUSH     DS                  ; Diese Register müssen
            PUSH     ES                  ; immer gesichert wer-
            PUSH     SI                  ; den, falls sie UDF sie
            PUSH     DI                  ; ändert
            ;
            ; ggf. weitere Register sichern (PUSH)
            ;

            ;
            ; Anwendungs-abhängiger Code
            ;
```

```
            ;
            ; ggf. oben gesicherte Register zurückholen (POP)
            ;
            POP     ES                  ; "Standard-Register"
            POP     DS                  ; zurückholen
            POP     BP

            ;
            ; ggf. Rückgabewert auf den Stack (PUSH)
            ;
            CALL    <Rückgabe-Funktion>
            ADD     SP,<n>              ; Stack bereinigen,
                                        ; <n> entspricht der
                                        ; Anzahl in Bytes des
                                        ; zuvor auf den
                                        ; Stack gebrachten
                                        ; Wertes.

            RET                         ; Rücksprung nach
                                        ; Clipper

<UDF-Name> ENDP

<CS-Name>  ENDS

            END                         ; Ende des Quellcodes
```

9.2 Verwendung der Funktion __PARINFO()

Wie bereits gesagt, gibt `__PARINFO()` Informationen über die Anzahl und die Typen der übergebenen Parameter zurück.

9.2.1 Bestimmung der Parameter-Anzahl

Zur Bestimmung der Parameter-Anzahl muß der Aufruf `__PARINFO(0)` nachgebildet werden. Dazu dient folgender Assembler-Code:

```
XOR     AX,AX           ; AX = 0
PUSH    AX              ; Parameter 0 auf den
                        ; Stack
CALL    __PARINFO       ; Parameterzahl -> AX
ADD     SP,2            ; Stack bereinigen
```

9.2.2 Bestimmung der Parameter-Typen

Der Parameter-Typ wird mit `__PARINFO()` bestimmt, indem man vor dem Aufruf die Nummer des gewünschten Parameters auf den Stack bringt, wie folgendes Codebeispiel zeigt:

```
;
; Typ des zweiten Parameters lesen
;
MOV     AX,2            ; Parameter-Nummer -> AX
PUSH    AX              ; auf den Stack
CALL    __PARINFO       ; Typkenner -> AX
ADD     SP,2            ; Stack bereinigen
```

Die Tabelle der möglichen Typkenner finden Sie im vorigen Kapitel.

9.3 Verwendung der Funktion __PARINFA()

Die Funktion `__PARINFA()` erfüllt ähnliche Aufgaben wie `__PARINFO()` jedoch bezogen auf Array-Variablen.

9.3.1 Bestimmung der Elementzahl eines Arrays

Zur Bestimmung der Parameter-Anzahl muß der Aufruf `__PARINFA(<pos>,0)` nachgebildet werden. Dazu dient folgender Assembler-Code:

```
XOR     AX,AX           ; AX = 0
PUSH    AX              ; 2. Parameter zuerst
MOV     AX,<pos>
PUSH    AX              ; dann <pos>
CALL    __PARINFA       ; Anzahl -> AX
ADD     SP,4            ; Stack bereinigen
```

Für <pos> muß jeweils der Wert eingesetzt werden, der die Position des Array-Namens beim Aufruf der Assmebler-UDF aus Clipper in der Parameterliste kennzeichnet.

9.3.2 Bestimmung des Typs eines Array-Elementes

Zur Bestimmung des Typs eines Array-Elementes muß der Aufruf __PARINFA(<pos>,<index>) nachgebildet werden. Dazu dient folgender Assembler-Code:

```
MOV     AX,<index>      ;
PUSH    AX              ; Index -> Stack
MOV     AX,<pos>
PUSH    AX              ; Position -> Stack
CALL    __PARINFA       ; Typkenner -> AX
ADD     SP,4            ; Stack bereinigen
```

9.4 Übernahme von Parametern

Zur Übernahme unterschiedlicher Parameter-Typen bietet Clipper die bereits erwähnten Standard-Funktionen, die stets nach folgendem Schema verwendet werden:

1. Nummer des gewünschten Parameters auf den Stack bringen
2. Aufruf der dem Parameter-Typ entsprechenden Übernahme-Funktion
3. Der Wert des Parameters oder ein Zeiger auf den Wert (bei Strings) ist in Registern enthalten (siehe Tabelle weiter unten)
4. Stack bereinigen

Hier ein Beispiel, wie der zweite Parameter gelesen wird (er enthält einen String):

```
MOV     AX,2            ; Parameter-Nr. 2
PUSH    AX              ; auf den Stack
CALL    __PARC          ; Character-String
ADD     SP,2            ; Stack bereinigen
;
; DX:AX enthalten Zeiger auf den String
;
; DX := Segment
; AX := Offset
;
```

In der nachfolgenden Tabelle finden Sie eine Zusammenstellung der Funktionen zur Parameter-Übernahme und in welchen Prozessor-Registern nach dem Aufruf der übernommene Wert enthalten ist.

Clipper-Datentyp	C-Typ	Funktion	Register
String	`char*`	`__PARC`	`DX:AX` (Zeiger, Segment:Offset)
Numerisch	`int`	`__PARCLEN`	`AX` (Länge eines Strings)
Numerisch	`int`	`__PARCSIZ`	`AX` (Speicherbedarf eines Strings)
Datum	`char*`	`__PARDS`	`DX:AX` (Zeiger, Segment:Offset)
Logisch	`int`	`__PARL`	`AX`
Numerisch	`double*`	`__PARND`	`DX:AX` (Zeiger, Segment:Offset)
Numerisch	`int`	`__PARNI`	`AX`
Numerisch	`long`	`__PARNL`	`DX:AX`

Die Funktionen `__PARC()`, `_PARDS` und `__PARND()` geben einen Zeiger zurück, d.h. die Adresse, ab der die Zeichenkette im Speicher enthalten ist. Beim Datum-Typ erfolgt durch `__PARDS()` eine Umwandlung in einen String, da Clipper intern ein schwer handhabbares Datumformat verwendet. Zeichenketten werden (wie in C allgemein üblich) mit einem Null-Byte abgeschlossen und enthalten keine Längeninformation.

Die oben angegebenen Aufrufe zur Parameter-Übernahme lassen sich sowohl für einzelne Parameter, als auch für Array-Elemente benutzen. Bei einem einzelnen Parameter wird (wie im obigen Beispiel gezeigt) die Positions-Nummer des Parameters in der Parameter-Liste auf den Stack gebracht und dann die entsprechende Übernahme-Funktion aufgerufen. Bei einem Array-Element muß

nach der Positions-Nummer zusätzlich der Index des Elementes auf den Stack gebracht werden.

Beachten Sie, daß nach Aufruf der Übernahme-Routine *unbedingt* der Stack wieder bereinigt werden muß. Bei Einzelparametern addieren Sie dazu den Wert 2 und bei Array-Parametern den Wert 4 zum Register `SP`.

Die Aufrufe `__PARCLEN()` und `__PARCSIZ()` nehmen eine gewisse Sonderstellung ein. `__PARCLEN()` gibt die Länge eines Strings zurück. Dabei wird das abschließende Null-Zeichen nicht mitgezählt. Sind allerdings im String selbst Null-Zeichen enthalten, werden Sie wie jedes andere Zeichen mitgezählt.

`__PARCSIZ()` dagegen gibt den gesamten von einem String benötigten Speicherplatz zurück, also einschließlich aller eingeschlossenen und des abschließenden Null-Zeichens.

Wichtig ist, daß bei Strings der Zeiger auf die angesprochene Zeichenkette übergeben wird. Dadurch bedingt, wirken sich Änderungen, die die UDF an der Zeichenkette vornimmt, auch im aufrufenden Clipper-Programm aus.

Wenn Sie eine Zeichenkette zurückgeben möchten, dann müssen Sie dazu die Rückgabefunktion `__RETC()` verwenden. Hier muß besonders darauf geachtet werden, daß die Zeichenkette mit einem Null-Byte abgeschlossen wird.

Wichtig: Die `__PARx()`-Funktionen sichern nur die Register `SP`, `BP`, `DS`, `SS`, `CS` und `IP`. Wenn der Inhalt anderer Register von Bedeutung ist, so müssen diese vor den Funktionsaufrufen auf dem Stack gesichert und anschließend wieder zurückgeholt werden.

9.5 Aufruf von Assembler-UDFs aus einem Clipper-Programm

Prinzipiell gibt es drei Möglichkeiten, eine Assembler-UDF aus einem Clipper-Programm aufzurufen:

1. Als Funktion, z.B.:

   ```
   X = udf(p1, ..., pn)
   ```

   Bei dieser Aufruf-Form ist die Rückgabe eines Wertes möglich.

2. Als Prozedur, z.B.:

```
DO udf WITH p1, ..., pn
```

Hier kann kein Wert zurückgegeben werden, d.h. die UDF muß mit einem Aufruf von __RET() beendet werden.

3. Mit einem CALL-Aufruf, z.B.:

```
CALL udf WITH p1, ..., pn
```

Diese Form des Aufrufes sollte nicht verwendet werden, da hier keine Werteübergabe über die eingebaute "Software-Schnittstelle" erfolgt, d.h. der Aufruf kann bei späteren Clipper-Versionen nicht mehr ordnungsgemäß arbeiten. CALL-Aufrufe verwenden außerdem den Clipper-internen Daten- und Stack-Bereich, so daß leicht die Gefahr besteht, daß interne Werte geändert werden, die dann zu einem "Programm-Absturz" führen.

CALL-Anweisungen können gut verwendet werden, um einige Clipper-interne Funktionen aus einem Programm heraus zu verwenden, wie an anderer Stelle in diesem Buch gezeigt wird.

9.6 Universelle Makro-Datei für Assembler-UDFs

Wie beim allgemeinen Aufbau einer Assembler-UDF beschrieben, müssen bestimmte Funktionen als extern deklariert und besondere Konventionen bei der Parameterübernahme und -rückgabe eingehalten werden. Außerdem gibt es für die Datentypen entsprechende Typkenner. Zur Vereinfachung beim Erstellen eines Assemblerprogramms kann die nachstehend gezeigte Makro-Datei dienen, die entsprechende Definitionen enthält und die Sie mit der Anweisung

```
INCLUDE   EXTENDA.MAC
```

am Anfang des Quellprogramms einlesen können.

Diese Datei wurde von Ralph Davis, Tom Rettig Associates geschrieben und als Public Domain Software zur allgemeinen Verwendung freigegeben. Vielen Dank, Ralph und Tom!

```
; Dateiname: EXTENDA.MAC
; Programm.: Clipper Extended Library
; Autor....: Ralph Davis, Ergänzungen Günther Daubach
; Datum....: 15. Mai 1988
; Anmerkung: Freigegeben als Public Domain Programm von
;            Tom Rettig Associates
;            Clipper ist Warenzeichen von Nantucket.
; Bemerkung: Assembler Makros für die Clipper "Software-
;            Schnittstelle". Die Maykros berücksichtigen keine
;            Array-Elemente
;
;
          EXTRN    __PARINFO:FAR
          EXTRN    __PARC:FAR
          EXTRN    __PARNI:FAR
          EXTRN    __PARNL:FAR
          EXTRN    __PARND:FAR
          EXTRN    __PARDS:FAR
          EXTRN    __PARL:FAR

          EXTRN    __RETC:FAR
          EXTRN    __RETNI:FAR
          EXTRN    __RETNL:FAR
          EXTRN    __RETND:FAR
          EXTRN    __RETDS:FAR
          EXTRN    __RETL:FAR

; Typ-Definitionen (entspr. EXTEND.H für C-UDFs)

UNDEF     EQU       0
CHARACTER EQU       1
NUMERIC   EQU       2
LOGICAL   EQU       4
DATE      EQU       8
ALIAS     EQU      16

TRUE      EQU       1
FALSE     EQU       0
```

```
; Parameter-Anzahl nach AX lesen

GET_PCOUNT MACRO
           XOR     AX,AX
           PUSH    AX
           CALL    __PARINFO
           ADD     SP,2
           ENDM

; Typ des angegebenen Parameters N nach AX lesen

GET_PTYPE  MACRO   N
           MOV     AX,N
           PUSH    AX
           CALL    __PARINFO
           ADD     SP,2
           ENDM

; Zeiger auf den angegebenen String-Parameter N nach
; AX:BX lesen

GET_CHAR   MACRO   N
           MOV     AX,N
           PUSH    AX
           CALL    __PARC
           ADD     SP,2
           MOV     BX,AX
           MOV     AX,DX
           ENDM

; Angegebenen Integer-Parameter N nach AX lesen

GET_INT    MACRO   N
           MOV     AX,N
           PUSH    AX
           CALL    __PARNI
           ADD     SP,2
           ENDM
```

```
; Angegebenen Long-Parameter N nach AX:BX lesen

GET_LONG   MACRO    N
           MOV      AX,N
           PUSH     AX
           CALL     __PARNL
           ADD      SP,2
           MOV      BX,AX
           MOV      AX,DX
           ENDM

; Angegebenen Double-Parameter N nach AX:BX:CX:DX lesen

GET_DBL    MACRO    N
           MOV      AX,N
           PUSH     AX
           CALL     __PARND
           ADD      SP,2
           MOV      ES,DX
           MOV      SI,AX
           MOV      AX,ES:[SI]
           MOV      BX,ES:[SI+2]
           MOV      CX,ES:[SI+4]
           MOV      DX,ES:[SI+6]
           ENDM

; Zeiger auf den angegebenen Datum-String-Parameter N nach
; AX:BX lesen

GET_DATESTR MACRO   N
           MOV      AX,N
           PUSH     AX
           CALL     __PARDS
           ADD      SP,2
           MOV      BX,AX
           MOV      AX,DX
           ENDM
```

```
; Angegebenen logischen-Parameter N nach AX lesen
; 0 := .F.
; 1 := .T.

GET_LOGICAL MACRO   N
            MOV     AX,N
            PUSH    AX
            CALL    __PARL
            ADD     SP,2
            ENDM

; String-Zeiger aus den Registern REG1:REG2 zurückgeben

RET_CHAR    MACRO   REG1,REG2
            IRP     X,<REG1,REG2>
            PUSH    X
            CALL    __RETC
            ADD     SP,4
            ENDM

; Ganzzahl aus Register REG1 zurückgeben

RET_INT     MACRO   REG1
            PUSH    REG1
            CALL    __RETNI
            ADD     SP,2
            ENDM

; Long-Integer aus Registern REG1:REG2 zurückgeben

RET_LONG    MACRO   REG1,REG2
            IRP     X,<REG1,REG2>
            PUSH    X
            CALL    __RETNL
            ADD     SP,4
            ENDM
```

```
; 8-Byte Gleitpunktzahl aus den Registern REG1:REG2:REG3:REG4
; zurückgeben

RET_DBL     MACRO     REG1,REG2,REG3,REG4
            IRP       X,<REG1,REG2,REG3,REG4>
            PUSH      X
            CALL      __RETND
            ADD       SP,8
            ENDM

; Zeiger auf Datum-String aus den Registern REG1:REG2
; zurückgeben

RET_DATESTR MACRO     REG1,REG2
            IRP       X,<REG1,REG2>
            PUSH      X
            CALL      __RETDS
            ADD       SP,4
            ENDM

; Logisch wahr (1) oder falsch (0) aus REG1 zurückgeben

RET_LOGICAL MACRO
            PUSH      REG1
            CALL      __RETL
            ADD       SP,2
            ENDM

SAV_REGS    MACRO                          ; "Standard-Register"
                                           ; sichern
            PUSH      BP
            MOV       BP,SP
            PUSH      BX
            PUSH      CX
            PUSH      DX
            PUSH      SI
            PUSH      DI
            PUSH      DS
            PUSH      ES
            ENDM
```

```
REST_REGS  MACRO                          ; "Standard-Register"
                                          ; wiederherstellen
           POP     ES
           POP     DS
           POP     DI
           POP     SI
           POP     DX
           POP     CX
           POP     BX
           POP     BP
           ENDM
```

Durch Einlesen dieser Datei in Ihre Assemblerprogramme ersparen Sie sich das Deklarieren der Übergabefunktionen als EXTRN und können außerdem den Programmecode für die Übernahme und Rückgabe von Parametern vereinfachen, indem Sie die hier definierten Makros verwenden.

So lesen Sie die Anzahl der übergebenen Parameter mit dem Makro-Aufruf

```
GET_PCOUNT
```

in das Register AX ein.

Der Makro-Aufruf

```
GET_PTYPE 2
```

bewirkt, daß der Typkenner des zweiten Parameters in das Register AX gelesen wird.

Da auch die Typkenner vordefiniert sind, können Sie Namen statt Zahlen verwenden. Das folgende Programmbeispiel verwendet mehrere Makro-Aufrufe. Die damit definierte Funktion ISCHAR() gibt .T. zurück, wenn der übergebene Parameter vom Typ String ist.

```
            PUBLIC   ISCHAR

            INCLUDE  EXTENDA.MAC

CODESEG     SEGMENT  'CODE'
            ASSUME   CS:CODESEG

ISCHAR      PROC     FAR

            SAV_REGS                  ; "Standard-Reg." sichern
            GET_PTYPE 1               ; Typ des ersten Parameters

            TEST     AX,CHARACTER     ; String oder Memo-String ?
            JZ       NOCHAR
            MOV      AX,TRUE          ; ist String, also wahr
            JMP      FINITO
NOCHAR:     MOV      AX,FALSE         ; Kein String, also falsch
FINITO:     RET_LOGICAL AX            ; logischen Wert zurückgeben

            REST_REGS                 ; Register wiederherstellen
            RET                       ; Rücksprung

ISACHR      ENDP
CODESEG     ENDS
            END
```

9.7 Sammlung nützlicher UDFs in Assemblersprache

In diesem Kapitel finden Sie UDFs in Assemblersprache, die nützliche Funktionen enthalten. Sie können diese in Ihre Clipper-Applikationen einbinden. Die Quellprogramme sind auch auf der beiliegenden Diskette gespeichert. Die assemblierten Routinen wurden in die ebenfalls auf der Diskette enthaltenen Bibliothek-Datei `TT87.LIB` aufgenommen. Wenn Sie keine Änderungen an den UDFs vornehmen wollen, können Sie `TT87.LIB` direkt einbinden. Zur Änderung der UDFs benötigen Sie neben einem Text-Editor auch einen Assembler (empfohlen wird MASM 4.0 oder eine höhere Version von Microsoft).

Alle UDFs setzen die Existenz der im vorigen Kapitel beschriebenen Datei `EXTENDA.MAC` voraus.

9.7.1 Abfragen, ob der Cursor eingeschaltet ist

```
; ISCURS.ASM
;
; von Günther Daubach
;
; SYNTAX:      logVar = ISCURS()
;
; RÜCKGABE:    .T., wenn der Cursor sichtbar ist, sonst .F.
;
        PUBLIC      ISCURS
        INCLUDE     EXTENDA.MAC
CODESEG SEGMENT     'CODE'
        ASSUME      CS:CODE

ISCURS  PROC  FAR
        SAV_REGS                    ; "Standard-Register" sichern
        CALL  __GETCTYP             ; interne Clipper-Funktion
        RET_LOGICAL AX              ; Cursor-Zustand zurückgeben
        REST_REGS                   ; "Standard-Register" zurück
        RET
ISCURS  ENDP

CODESEG ENDS
        END
```

Mit dieser UDF kann abgefragt werden, ob der Cursor aktuell sichtbar ist oder nicht. Das ist z.B. hilfreich, wenn in HELP.PRG oder in einer anderen über Tastendruck aufgerufenen Prozedur der Cursor ausgeschaltet werden soll und vor Verlassen der Prozedur wieder den ursprünglichen Zustand annehmen soll.

9.7.2 Töne bestimmter Höhe und Dauer erzeugen

```
; SOUND.ASM
;
; von Günther Daubach
;
; SYNTAX:     logVar = SOUND(<Frequenz>,<Dauer>)
;
; RÜCKGABE:   .T., wenn die angegebenen Parameter korrekt sind,
;             sonst .F.
;
; FUNKTION:   Erzeugt einen Ton der angegebenen Frequenz in
;             Hertz mit der angegebenen Dauer. Die Dauer ist
;             nicht geeicht und hängt vom Systemtakt des Com-
;             puters ab.
;
            PUBLIC SOUND

            INCLUDE EXTENDA.MAC

PORTB       EQU     61H

CODESEG     SEGMENT 'CODE'
            ASSUME CS:CODESEG

SOUND       PROC    FAR
            SAV_REGS            ; "Standard-Register" retten
            GET_PCOUNT          ; Anzahl Parameter -> AX   (
            CMP     AX,2        ; 2 Parameter?             (1
            JZ      OK          ; ja                       (
FEHL1:      MOV     AX,FALSE    ; .F. zurückgeben
            JMP SHORT FEHLER
OK:         GET_PTYPE 1         ; Typ des 1. Parameters    (
            CMP     AX,NUMERIC  ; numerisch?               (
            JNZ     FEHL1       ; nein, Fehler             (2
            GET_PTYPE 2         ; Type des 2. Parameters   (
```

```
            CMP     AX,NUMERIC      ; numerisch?                (
            JNZ     FEHL1           ; nein, Fehler              (
            GET_INT 2               ; Dauer -> AX
            PUSH    AX              ; Dauer sichern
            GET_INT 1               ; Frequenz -> AX
            CMP     AX,18           ; prüfen, ob > 18           (3
            JG      OK1
            POP     AX              ; Stack "aufräumen"
            JMP SHORT FEHL1         ; und Fehlerrückgabe
OK1:        MOV     BX,AX           ; Frequenz -> BX
            XOR     AX,AX           ; AX = 0
            MOV     DX,12H
            DIV     BX              ; Periode berechnen
            MOV     BX,AX           ; Periode -> BX
            MOV     AL,10110110B    ; Timer2 initialisieren
            OUT     43H,AL
            MOV     AX,BX           ; Periode -> AX
            OUT     42H,AL          ; LSB -> Timer2
            MOV     AL,AH           ; MSB -> AL
            OUT     42H,AL          ; MSB -> Timer2
            IN      AL,PORTB        ; Port B lesen
            OR      AL,3            ; Bits 0 und 1 ein
            OUT     PORTB,AL        ; Lautsprecher ein
            POP     CX              ; Dauer -> CX
WAIT:       PUSH    CX              ; Zähler sichern
            MOV     CX,1000         ; Festwert für
WAIT1:      LOOP    WAIT1           ; innere Schleife
            POP     CX              ; Zähler lesen
            LOOP    WAIT            ; äußere Schleife
            IN      AL,PORTB        ; Port B lesen
            AND     AL,11111100B    ; Bits 0 und 1 aus
            OUT     PORTB,AL        ; Lautsprecher aus
            MOV     AL,TRUE         ; .T. zurückgeben
FEHLER:     REST_REGS               ; Register wiederherstellen
            RET_LOGICAL AX          ; Status zurückgeben        (4
            RET                     ; fertig

SOUND       ENDP
CODESEG     ENDS
            END
```

Diese UDF können Sie zum Erzeugen unterschiedlicher Signaltöne verwenden, um z.B. verschiedene Programmsituationen akustisch unterschiedlich zu kennzeichnen. Wenn Sie wollen, können Sie sich damit natürlich auch in Clipper damit ein Musikprogramm schreiben.

Hier einige Hinweise zum Programm (sie finden im Programmtext entsprechende Kennziffern):

1) Durch die Prüfung, ob die richtige Anzahl Parameter übergeben wurde, kann die UDF "betriebssicher" (um nicht zu sagen "programmierersicher") gemacht werden. Dadurch wird verhindert, daß bei falscher Anwendung der Funktion unsinnige Ergebnisse liefert oder gar das Programm zum Absturz bringt.

2) Auch die Prüfung, ob die übergebenen Parameter die verlangten Typen haben, ist für die "Betriebssicherheit" von entscheidender Bedeutung.

3) Der dritte, wichtige Faktor für die "Betriebssicherheit" einer UDF ist die Prüfung, ob die übergebenen Parameter im zulässigen Wertebereich liegen. In diesem Fall muß die Frequenz über 18 liegen, da sonst die spätere Division (DIV BX) zu einem Teiler-Überlauf Interrupt führt.

4) Obwohl diese Funktion kein Ergebnis liefern muß, da sie ja als gewünschte "Rückgabe" einen Ton im Lautsprecher erzeugt, wird hier .T. oder .F. zurückgegeben, abhängig davon, ob die übergebenen Parameter korrekt waren oder nicht. Dadurch erreicht man, daß das aufrufende Programm auch bei eigentlich "rückgabelosen" Funktionen prüfen kann, ob die Funktion fehlerfrei ausgeführt wurde.

In Clipper ist ab Version Sommer '87 eine Funktion TONE() enthalten, die in etwa der hier beschriebenen Routine entspricht, allerdings können mit SOUND() kürzere Tondauern als mit TONE() erzielt werden.

9.7.3 Form des Cursors verändern

```
; CURSFORM.ASM
; von Günther Daubach
;
; SYNTAX:     logVar = CURSFORM(<obere Zeile>,<untere Zeile>)
;
; RÜCKGABE:   .T., wenn die angegebenen Parameter korrekt sind,
;             sonst .F.
;
; FUNKTION:   Ändert die Form des Cursors, wie durch die beiden
;             Parameter angegeben (siehe auch IBM-BIOS Listing)
;
            PUBLIC CURSFORM
            INCLUDE EXTENDA.MAC

CODESEG     SEGMENT 'CODE'
            ASSUME CS:CODESEG

CURSFORM    PROC   FAR
            SAV_REGS            ; "Standard-Register" retten
            GET_PCOUNT          ; Anzahl Parameter -> AX
            CMP    AX,2         ; 2 Parameter?
            JZ     OK           ; ja
FEHL1:      MOV    AX,FALSE     ; .F. zurückgeben
            JMP SHORT FEHLER
OK:         GET_PTYPE 1         ; Typ des 1. Parameters
            CMP    AX,NUMERIC   ; numerisch?
            JNZ    FEHL1        ; nein, Fehler
            GET_PTYPE 2         ; Type des 2. Parameters
            CMP    AX,NUMERIC   ; numerisch?
            JNZ    FEHL1        ; nein, Fehler
            GET_INT 1           ; obere Cursorzeile
            MOV    CH,AL        ; -> CH
            PUSH   CX           ; sichern
            GET_INT 2           ; untere Cursorzeile
            POP    CX
            MOV    CL,AL        ; -> CL
            MOV    AH,1         ; BIOS Interrupt 10H,
            INT    10H          ; Cursorform setzen
            MOV    AX,TRUE      ; .T. zurückgeben
```

```
FEHLER:                         ; AX enthält .F.
            REST_REGS           ; Register wiederherstellen
            RET_LOGICAL AX      ; Status zurückgeben
            RET                 ; fertig
CURSFORM    ENDP
CODESEG     ENDS
            END
```

Diese Funktion erlaubt es, die Form des Cursors zu verändern. Dazu erwartet die Funktion die Parameter `<obere Zeile>` und `<untere Zeile>`. Diese Parameter bestimmen, in welcher Rasterzeile einer Textzeile der Cursor beginnt und wo er endet. Die zulässigen Werte hängen vom verwendeten Bildschirm-Adapter ab. Normalerweise sind bei Farb-Adaptern Werte zwischen 0 und 8 und beim Monochrom-Text-Adapter Werte zwichen 0 und 13 möglich.

Bei einigen Computersystemen können hier andere Angaben zutreffen. Beachten Sie, daß der Wert `<obere Zeile>` normalerweise kleiner als `<untere Zeile>` sein soll. Durch Vertauschen der Werte erhalten Sie einen "geteilten" Cursor, d.h., es erscheinen zwei blinkende Linien.

Sie können den Cursor mit dieser Funktion auch ganz ausschalten. Dazu geben Sie für `<obere Zeile>` einen Wert an, der größer ist als der Maximalwert (9 bei Farb- und 14 bei Monochrom-Adaptern) und setzen für `<untere Zeile>` den Wert 0 ein. Einfacher geht das aber mit den "eingebauten" Clipper-Kommandos `SET CURSOR OFF/ON`.

9.7.4 Bestimmen des aktiven Laufwerks

```
; DRIVE.ASM
;
; von Günther Daubach
;
; SYNTAX:     charVar = DRIVE()
;
; RÜCKGABE:   Kenner des aktuellen Laufwerks
            PUBLIC DRIVE
            INCLUDE EXTENDA.MAC

CODESEG     SEGMENT 'CODE'
            ASSUME CS:CODESEG
```

```
DRIVE       PROC   FAR
            JMP SHORT CONT        ; "Umleitung" über den
                                  ; String-Speicher
LAUFW       DB     (?),0          ; Platz für Rückgabe-String
CONT:       MOV    AH,19H         ; Funktionsnummer für
            INT    21H            ; DOS-Interrupt 21H
            ADD    AL,'A'         ; Korrektur auf ASCII-Code
            MOV    CS:LAUFW,AL    ; Zeichen in String-Speicher
                                  ; String-Offset -> AX
            MOV    AX,OFFSET CS:LAUFW
                                  ; String-Segment ist in CS
            RET_CHAR CS,AX        ; String-Rückgabe
            RET                   ; fertig

DRIVE       ENDP
CODESEG     ENDS
            END
```

Diese UDF liefert den Kenner des aktuellen Laufwerkes von DOS zurück. Beachten Sie, daß ggf. in Clipper enthaltene Anweisunfen `SET DEFAULT TO` das Ergebnis dieser Funktion nicht beeinflussen. Es ist kein Parameter beim Aufruf erforderlich. Da durch die Funktion die "Standard-Register" nicht verändert werden, ist es nicht nötig, diese zu sichern.

Beachten Sie, wie der temporäre Speicher für die Zeichenkette mit dem Laufwerkkenner innerhalb des Codesegmentes angeordnet ist. Der Sprung über diesen Speicher ist nötig, damit er im `PROC ... ENDP` Block angelegt werden kann. Wenn auch unscheinbar, so ist sie doch enorm wichtig: die 0 in `DB(?),0`, die das Nullbyte zur Beendigung der Zeichenkette definiert.

9.7.5 Aktuellen Pfad abfragen

```
; WHATDIR.ASM
;
; von Günther Daubach
;
; SYNTAX:     charVar = WHATDIR(["<Laufwerk>"])
;
; RÜCKGABE:   String mit aktuellem Pfad (mit vorangestelltem
;             "\"), wenn gültiges Laufwerk und gültiger
;             Parameter-Type, sonst "?"
```

```
;
; FUNKTION:  Fragt den aktuellen Pfad des angegebenen Lauf-
;            werkes ab. Wird kein Parameter angegeben, dann
;            wird der Pfad des aktuellen Laufwerkes gelesen.
;
            PUBLIC WHATDIR
            INCLUDE EXTENDA.MAC

CODESEG     SEGMENT 'CODE'
            ASSUME CS:CODESEG
WHATDIR     PROC   FAR
            JMP SHORT CONT          ; "Umleitung" über den
                                    ; String-Speicher
DIRBUFF     DB     64 DUP (?),0     ; Platz für Rückgabe-String
CONT:       SAV_REGS                ; "Standard-Register" retten
            GET_PCOUNT              ; Anzahl Parameter lesen
            AND    AX,AX            ; Flags setzen
            JZ     OHNEPARAM        ; 0 = keine Parameter
            GET_PTYPE               ; Parameter-Typ lesen
            DEC    AL               ; 1 - 1 = 0
            JNZ    FEHLER           ; ist nicht char, Fehler
            GET_CHAR 1              ; Zeiger auf String -> AX:BX
            PUSH   DS               ; DS retten und auf String
            MOV    DS,AX            ; Segment setzen
            MOV    DL,DS:[BX]       ; erstes Zeichen von String
                                    ; nach DL (Rest unwichtig)
            POP    DS               ; DS wiederherstellen
            AND    DL,0DFH          ; Großbuchstaben erzwingen
            SUB    DL,'A'-1         ; in Laufwerksnummer umwan-
                                    ; deln (A: = 1, B: = 2 usw.)
            JMP SHORT CONT1
OHNEPARAM:  XOR    DL,DL            ; kein Parameter, also aktu-
                                    ; elles Laufw. annehmen (0)
CONT1:      PUSH   DS               ; DS retten
            MOV    AH,'\'           ; String mit '\' vorbelegen
            MOV    CS:DIRBUFF,AH
            MOV    AH,47H           ; Funktion 47H (Dir. lesen)
            PUSH   CS               ; Puffer-Segment (CS:) nach
            POP    DS               ; DS übertragen
                                    ; Offset (nach '\') -> SI
            MOV    SI,OFFSET DIRBUFF+1
            INT    21H              ; DOS Interrupt 21H
```

```
            POP     DS              ; Clipper Datensegment
                                    ; wiederherstellen
            JNC     KEINFEHLER      ; kein Fehler von INT 21H
                                    ; meldet
FEHLER:     MOV     AH,'?'          ; bei Fehler '?' als Rück-
            MOV     CS:DIRBUFF,AH   ; gabe in den Puffer schrei-
            XOR     AH,AH           ; ben und Nullbyte
                                    ; nicht vergessen!
            MOV     CS:DIRBUFF+1,AH
KEINFEHLER:
            REST_REGS               ; "Standardregister" zurück
                                    ; String über CS:AX zurück-
                                    ; geben
            MOV     AX,OFFSET CS:DIRBUFF
            RET_CHAR CS,AX
            RET                     ; fertig
WHATDIR     ENDP
CODESEG     ENDS
            END
```

Diese Funktion gibt den aktuellen Pfad eines Laufwerks von DOS zurück. Eventuelle Anweisungen `SET PATH TO...` in Clipper beeinflussen die Rückgabe der Funktion nicht. Sie erkennen hier, wie eine UDF so ausgelegt werden kann, daß eine variable Anzahl von Parametern möglich ist. Wird kein Parameter übergeben, nimmt die Funktion das aktuelle Laufwerk an, sonst das Laufwerk, dessen Kenner als Parameter übergeben wurde.

Auch hier werden verschiedene Fehlersituationen abgefangen: falscher Parameter-Typ oder DOS-Fehler beim Lesen des aktuellen Pfades (wenn z.B. ein nicht vorhandenes Laufwerk angesprochen wird). In solchen Fällen wird statt der Pfad-Angabe ein Fragezeichen zurückgegeben.

9.7.6 Laufwerk prüfen

```
; ISDRIVE.ASM
;
; von Günther Daubach
;
; SYNTAX:   logVar = ISDRIVE("<Laufwerk>")
;
```

```
; RÜCKGABE: .T., wenn das angegebene Laufwerk bereit ist
;          .F., wenn das angegebene Laufwerk nicht bereit
;               oder nicht vorhanden ist oder wenn ein un-
;               gültiger Parameter übergeben wurde.
          PUBLIC  ISDRIVE
          INCLUDE EXTENDA.MAC
CODESEG   SEGMENT ' CODE'
          ASSUME  CS:CODESEG
ISDRIVE   PROC    FAR
          JMP     CONT          ; "Umleitung"
DBUFF     DB      512 DUP (0)   ; Sektor-Lesepuffer
CONT:     SAV_REGS
          GET_PCOUNT            ; Parameterzahl = 1 ?
          DEC     AL
          JNZ     FEHLER        ; nein, Fehler
          GET_PTYPE 1           ; Parameter-Typ char ?
          DEC     AL
          JZ      OK            ; ja, weiter
FEHLER:   MOV     AX,FALSE      ; .F. zurückgeben
          REST_REGS
          RET_LOGICAL AX
          RET
OK:       GET_CHAR 1            ; Laufwerk-Kenner lesen
          PUSH    DS            ; Clipper DS sichern
          MOV     DS,AX
          MOV     AL,DS:[BX]    ; Laufwerk -> AL
          AND     AL,0DFH       ; UPPER(<Laufwerk>)
          SUB     AL,'A'        ; umsetzen auf 0...
          ; DOS Interrupt 25H vorbereiten
          PUSH    CS
          POP     DS            ; DS = CS
          MOV     BX,OFFSET DBUFF ; Puffer-Offset
          MOV     CX,1          ; einen Sektor lesen
          XOR     DX,DX         ; Sektor 0
          INT     25H           ; DOS-Interrupt
;
;         die nachfolgende Anweisung ist nötig, um den
;         Stack zu "bereinigen", da INT 25H leider den
;         Stack nicht auf den Wert vor dem Aufruf zu-
;         rücksetzt - das ist kein sauberer Programmier-
;         stil, Microsoft !
```

```
            POP     AX
            POP     DS              ; Clipper DS zurückholen
            JC      FEHLER          ; CY = 1 - Fehler !
            MOV     AX,TRUE         ; Laufw. ok, .T. zurückgeben
            REST_REGS
            RET_LOGICAL AX
            RET
ISDRIVE     ENDP
CODESEG     ENDS
            END
```

Diese Funktion ermöglicht es, abzufragen, ob ein Laufwerk bereit ist. Das ist besonders dann wichtig, wenn mit Diskettenlaufwerken gearbeitet wird, um festzustellen, ob eine Diskette eingelegt ist. Mit dieser Funktion kann vermieden werden, daß DOS die Fehlermeldung "Nicht bereit..." anzeigt und damit die Bildschirm-Anzeige von Clipper zerstört.

Die Funktion stellt die Bereitschaft eines Laufwerks dadurch fest, daß versucht wird, einen Sektor direkt zu lesen. Bei dieser Lese-Operation der unteren Ebene wird keine DOS-Fehlermeldung erzeugt. Bei Computern der Typen IBM-AT und kompatiblen gibt es zwar einen Interrupt-Aufruf, der direkt die Bereitschaft eines Laufwerks meldet, allerdings steht dieser Aufruf auf Systemen vom Typ IBM-PC/XT nicht zur Verfügung, so daß Sie `ISDRIVE()` immer dann verwenden sollten, wenn Ihre Programme universell einsetzbar sein müssen.

9.7.7 Bildschirm mit Zeichen und Attribut füllen

```
; FILL.ASM
;
; von Günther Daubach
;
; SYNTAX:     logVar = FILL("<Zeichen>",<Attribut>,<logWert>)
;
; RÜCKGABE:   .T., wenn die angegebenen Parameter korrekt sind,
;             sonst .F.
;
; FUNKTION:   Füllt den Bildschirm mit dem angegebenen Zeichen
;             und verwendet das angegebene Attribut.
;             Ist <logWert> = .T., dann wird der ganze Schirm
;             gefüllt. Wenn <logWert> = .F., dann bleibt die
;             oberste Zeile unverändert.
;
```

```
BW_SEG      EQU     0B000H
COLOR_SEG   EQU     0B800H
            PUBLIC FILL
            INCLUDE EXTENDA.MAC
CODESEG     SEGMENT 'CODE'
            ASSUME CS:CODESEG
FILL        PROC    FAR
            JMP SHORT CONT
SCHNEE      DB      0               ; Merker für Farbbildschirm
DISP_SEG    DW      0               ; Bildspeicher-Segment
CONT:       SAV_REGS                ; "Standard-Register" retten
            GET_PCOUNT              ; Parameter-Anzahl = 2 ?
            CMP     AX,3
            JNZ     FEHLER
            GET_PTYPE 1             ; 1. Parameter char ?
            DEC     AX
            JNZ     FEHLER
            GET_PTYPE 2             ; 2. Parameter numerisch ?
            CMP     AX,NUMERIC
            JNZ     FEHLER
            GET_PTYPE 3             ; 3. Parameter logisch ?
            CMP     AX,LOGICAL
            JZ      INIT
FEHLER:     REST_REGS               ; Fehler - .F. zurückgeben
            MOV     AX,FALSE
            RET_LOGICAL
            RET
INIT:       INT     011H            ; BIOS Gerätestatus
            MOV     BX,BW_SEG       ; Monochrom annehmen
            AND     AX,30H
            CMP     AX,30H
            JE      INIT_1          ; ist monochrom
            MOV     BX,COLOR_SEG    ; ist Farbe
            MOV     SCHNEE,1        ; Merker f. Farbe
INIT_1:     MOV     CS:DISP_SEG,BX  ; Bildspeicher-Segment
            TEST    SCHNEE,0FFH     ; wenn Farbe, dann Anzeige
            JZ      FILL_2          ; während des Füllens aus-
            MOV     DX,03D8H        ; schalten, um Flimmern zu
            MOV     AL,021H         ; vermeiden
            OUT     DX,AL
FILL_2:     GET_LOGICAL 3           ; Kenner "Vollbildschirm"
            PUSH    AX              ; lesen und sichern
```

```
            GET_INT 2               ; Attribut lesen
            PUSH    AX              ; und sichern
            GET_CHAR 1              ; Zeiger auf Zeichen lesen
            PUSH    DS              ; Clipper DS sichern
            MOV     DS,AX
            MOV     AL,DS:[BX]      ; Zeichen ist in AL
            POP     DS              ; Clipper DS zurückholen
            POP     BX              ; Attribut -> BL
            MOV     AH,BL           ; Zeichen und Attribut in
                                    ; AH:AL
            MOV     ES,DISP_SEG     ; ES zeigt auf Bildspeicher
            POP     BX              ; Kenner Vollbildschirm
            DEC     BL              ; wenn voll, dann ist BL = 0
            JZ      FILL_5
            MOV     DI,160          ; ab dem 160-ten Byte begin-
                                    ; nen (2. Zeile)
            MOV     CX,1920         ; Zeichenzähler (24 * 80)
            JMP     FILL_4
FILL_5:     XOR     DI,DI           ; DI = 0
            MOV     CX,2000         ; Zeichenzähler (25 * 80)
FILL_4:     MOV     ES:[DI],AX      ; Zch./Attr. -> Bildspeicher
            INC     DI              ; "Zielzeiger" auf nächstes
            INC     DI              ; Wort im Bildspeicher
            LOOP    FILL_4          ; nochmal, wenn CX <> 0
            TEST    SCHNEE,0FFH     ; Farbe ?
            JZ      FILL_3          ; nein
            MOV     DX,03D8H        ; ja, Bildschirm wieder ein-
            MOV     AL,029H         ; schalten
            OUT     DX,AL
FILL_3:     REST_REGS               ; "Standardregister" holen
            MOV     AX,TRUE         ; .T. zurückgeben
            RET_LOGICAL AX
            RET                     ; fertig
FILL        ENDP
CODESEG     ENDS
            END
```

Diese Funktion ermöglicht es, den gesamten Bildschirm mit dem Zeichen zu füllen, das als erster Parameter engegeben wird. Dabei wird das als zweiter Parameter angegebene Attribut verwendet.

So füllt z.B. die Clipper-Anweisung

```
c = fill(chr(176),1,.T.)
```

den gesamten Farbbildschirm mit einem blauen Punktraster. Wichtig ist, daß diese Funktion nicht die in Clipper mit SET COLOR TO definierten Attribute beeinflußt.

Da in vielen Clipper-Programmen die oberste Bildschirmzeile (0) freigehalten wird, um die System-Meldungen anzuzeigen (SET SCOREBOARD ON), kann bei der Funktion mit dem dritten Parameter angegeben werden, ob die oberste Zeile auch gefüllt werden soll (.T.) oder ob sie nicht verändert werden darf (.F.).

Auch diese Funktion gibt einen logischen Wert zurück, der die Gültigkeit der übergebenen Parameter kennzeichnet.

Eine Fülle weiterer nützlicher Assembler-UDFs finden Sie in der von KRS Unternehmensberatung-EDV vertriebenen Super-Toolbox für den Clipper Compiler. Besonders leistungsfähige Routinen gestatten es auf einfache Weise Programme in Fenstertechnik zu erstellen, wobei die Routinen automatisch das Sichern und Wiederherstellen von Bildschirm-Inhalten ausführen, die durch Fenster überlagert werden. Auch nützliche Routinen, wie das ständige Einblenden der aktuellen Systemzeit und des Zustandes von Sondertasten, das Umschalten der Druckausgabe auf verschiedene Ports (auf Wunsch auch mit Zeichenkonvertierung) und vielseitige Funktionen zur direkten Beeinflussung der Bildschirm-Ausgabe und zur Bearbeitung von Texten und Textdateien sind enthalten.

9.8 Grafik_Funktionen

Die nachfolgend beschriebenen Assembler-UDFs zeigen, daß man in Clipper auch grafische Funktionen integrieren kann. Die UDFs sind für einen Bildschirm-Adapter geschrieben, der kompatibel zum IBM Color Grafik Adapter ist. Bei anderen Grafik-Adaptern müssen die Routinen entsprechend geändert werden. Da alle Grafik-Ausgaben über BIOS-Aufrufe erfolgen, ist das nicht sehr kompliziert. Dieses Verfahren hat jedoch den Nachteil, daß die Ausgabegeschwindigkeit nicht besonders hoch ist.

9.8.1 Bildschirm-Modus umschalten

```
; G_MODE.ASM
;
; von Günther Daubach
;
; Syntax:   logVar = G_MODE(<Modus>)
;
; Rückgabe: .T., wenn der angegebene Parameter korrekt ist,
;           sonst .F.
;
; Funktion: Schaltet den Bildschirm-Adapter in einen der fol-
;           genden Modi:
;
;           Modus  Auflösung
;             0    40 * 25    Text,   monochrom
;             1    40 * 25    Text,   Farbe
;             2    80 * 25    Text,   monochrom
;             3    80 * 25    Text,   Farbe
;             4    320 * 200  Grafik, Farbe
;             5    320 * 200  Grafik, monochrom
;             6    640 * 200  Grafik, monochrom
;
            PUBLIC    G_MODE
            INCLUDE   EXTENDA.MAC
CODESEG     SEGMENT   'CODE'
            ASSUME    CS:CODESEG, DS:CODESEG
G_MODE      PROC      FAR
            SAV_REGS              ; "Standard-Register retten
            GET_PCOUNT
            DEC       AL          ; Parameterzahl = 1 ?
```

```
            JNZ       FEHLER       ; nein, Fehler
            GET_PTYPE 1
            CMP       AL,NUMERIC   ; Parameter-Typ numerisch ?
            JZ        OK           ; ja, weitermachen
FEHLER:     MOV       AX,FALSE     ; .F. zurückgeben bei Fehler
            REST_REGS
            RET_LOGICAL AX
            RET
OK:         GET_INT 1              ; Parameter lesen
            CMP       AL,6         ; > 6 ?
            JA        FEHLER       ; ja, Fehler
            XOR       AH,AH        ; AH = 0
            INT       10H          ; BIOS-Funkt. "Modus setzen"
            REST_REGS              ; "Standard"-Register zurück
            MOV       AX,TRUE      ; .T. zurückgeben
            RET_LOGICAL AX
            RET                    ; fertig
G_MODE      ENDP
CODESEG     ENDS
            END
```

Diese Funktion muß *vor* Benutzung einer der nachfolgenden Grafik-Routinen aufgerufen werden, um in den gewünschten Grafik-Modus umzuschalten. Nach erfolgter Grafik-Anzeige muß mit dieser Funktion wieder in einen Text-Modus zurückgeschaltet werden, damit die "normale" Bildschirm-Ausgabe von Clipper wieder funktioniert. Hierzu sollten Sie Modus 2 oder 3 wählen, da die Bildschirm-Ausgaberoutinen auf dieses Anzeigeformat abgestimmt sind.

Wenn Sie nicht in einen Text-Modus umschalten, erscheinen die Bildschirm-Ausgaben von Clipper als Punktfolge, da Clipper direkt in den Bildschirm-Speicher schreibt. Sollten Sie die Umschaltung vergessen haben, brechen Sie das Programm am besten mit Alt-C und Q ab. Im Betriebssystem wählen Sie mit dem MODE-Kommando wieder einen entsprechenden Text-Modus.

`G_MODE()` löscht beim Aufruf gleichzeitig den Bildschirm.

9.8.2 Grafikschirm löschen

```
; G_CLS.ASM
;
; von Günther Daubach
;
; Syntax:   logVar = G_CLS()
;
; Rückgabe: .T.
;
; Funktion: Löscht den Grafikbildschirm
;
            PUBLIC G_CLS
            INCLUDE EXTENDA.MAC

CODESEG     SEGMENT 'CODE'
            ASSUME  CS:CODESEG, DS:CODESEG
G_CLS       PROC    FAR
            SAV_REGS
            MOV     AX,0B800H     ; Grafikspeicher-Segment
            MOV     ES,AX         ; -> ES
            MOV     CX,2000H      ; Zähler setzen
            XOR     AX,AX         ; Füllwert für Bildschirm = 0
            MOV     DI,AX         ; Bildschirm-Offset (0)
            CLD                   ; "vorwärts"
            REP     STOSW         ; Speicher mit [AX] füllen
            MOV     AX,TRUE       ; immer .T. zurückgeben
            REST_REGS
            RET_LOGICAL AX
            RET
G_CLS       ENDP
CODESEG     ENDS
            END
```

Diese Funktion löscht den Grafikschirm, indem der Bildschirm-Speicher mit 0 gefüllt wird.

9.8.3 Hintergrundfarbe setzen

```
; G_HGR.ASM
;
; von Günther Daubach
;
; Syntax:   logVar = G_HGR(<Farbe>)
;
; Rückgabe: .T., wenn der angegebene Parameter korrekt ist,
;           sonst .F.
;
; Funktion: Setzt die Hintergrund-Farbe des Grafikschirms
;
            PUBLIC G_HGR
            INCLUDE EXTENDA.MAC
CODESEG     SEGMENT 'CODE'
            ASSUME  CS:CODESEG, DS:CODESEG
G_HGR       PROC    FAR
            SAV_REGS
            GET_PCOUNT          ; Parameter-Zahl = 1 ?
            DEC     AL
            JNZ     FEHLER      ; nein, dann Fehler
            GET_PTYPE 1         ; Parameter numerisch ?
            CMP     AX,NUMERIC
            JZ      OK
FEHLER:     REST_REGS           ; .F. zurückgeben
            MOV     AX,FALSE
            RET_LOGICAL AX
            RET
OK:         GET_INT 1           ; Hintergrundfarbe lesen
            AND     AL,0FH      ; nur 0...15 zulassen
            XOR     BH,BH
            MOV     BL,AL
            MOV     AH,11       ; BIOS-Interrupt
            INT     10H         ; "Hintergund setzen"
            MOV     AX,TRUE     ;.T. zurückgeben
            REST_REGS
            RET_LOGICAL AX
            RET
G_HGR       ENDP
CODESEG     ENDS
            END
```

Diese Funktion erlaubt es, die Hintergrundfarbe des Grafikschirms auszuwählen (nur im Modus 4).

Erlaubt sind Farbwerte zwischen 0 und 15; es gilt folgende Zuordnung:

Farbwert	Farbe
0	schwarz
1	blau
2	grün
3	cyan
4	rot
5	magenta
6	braun
7	weiß
8	grau
9	hellblau
10	hellgrün
11	hellcyan
12	hellrot
13	hellmagenta
14	gelb
15	weiß, hohe Helligkeit

9.8.4 Farb-Palette auswählen

```
; G_PAL
;
; von Günther Daubach
;
; Syntax: logVar = G_PAL(<Nummer>)
;
; Rückgabe: .T., wenn der angegebene Parameter korrekt ist,
;           sonst .F.
;
; Funktion: Wählt eine FARB-Palette aus
;           Nummer = 0: Hintergrund/grün/rot/gelb
;           Nummer = 1: Hintergrund/cyan/magenta/weiß
```

```
            PUBLIC   G_PAL
            INCLUDE  EXTENDA.MAC
CODESEG     SEGMENT  'CODE'
            ASSUME   CS:CODESEG, DS:CODESEG
G_PAL       PROC     FAR
            SAV_REGS
            GET_PCOUNT            ; Parameterzahl = 1 ?
            DEC      AL
            JNZ      FEHLER       ; nein
            GET_PTYPE 1
            CMP      AX,NUMERIC   ; Parameter numerisch ?
            JZ       OK           ; ja
FEHLER:     REST_REGS
            MOV      AX,FALSE
            RET_LOGICAL AX
            RET
OK:         GET_INT 1             ; Paletten-Nummer lesen
            AND      AL,1         ; nur 0 und 1 zulassen
            MOV      BH,1
            MOV      BL,AL        ; BIOS-Interrupt
            MOV      AH,11        ; "Palette setzen"
            INT      10H
            MOV      AX,TRUE
            REST_REGS
            RET_LOGICAL AX
            RET
G_PAL       ENDP
CODESEG     ENDS
            END
```

Diese Funktion erlaubt die Auswahl zwischen zwei möglichen Farb-Paletten. Je nach angewählter Palette können auf dem Grafikschirm vier verschiedene Farben dargestellt werden. Dabei gilt folgende Zuordnung:

Farb-Nr.	Palette 0	Palette 1
0	Hintergrund	Hintergrund
1	grün	cyan
2	rot	magenta
3	braun	weiß

9.8.5 Punkt setzen/löschen

```
; G_DOT.ASM
;
; von Günther Daubach
;
; Syntax:   logVar = G_DOT(<x>,<y>,<Farbe>)
;
; Rückgabe: .T., wenn die angegebenen Parameter korrekt sind,
;           sonst .F.
;
; Funktion: Schreibt einen Punkt auf dem Grafikschirm, Farbe 0
;           löscht den Punkt, Farbe 1...3 setzt den Punkt in
;           der angegebenen Farbe.
;           Bei Farbwerten 128...131 wird das Farb-Attribut
;           des Punktes mit dem an dieser Stelle zuvor gespei-
;           cherten Farb-Attribut EXKLUSIV-ODER verknüpft.

            PUBLIC  G_DOT
            INCLUDE EXTENDA.MAC
CODESEG     SEGMENT 'CODE'
            ASSUME  CS:CODESEG, DS:CODESEG
G_DOT PROC  FAR
            SAV_REGS
            GET_PCOUNT
            CMP     AL,3            ; Parameterzahl <> 3 ?
            JNZ     FEHLER          ; ja, Fehler
            GET_PTYPE 1
            CMP     AL,NUMERIC      ; Typen numerisch ?
            JNZ     FEHLER
            GET_PTYPE 2
            CMP     AL,NUMERIC
            JNZ     FEHLER
            GET_PTYPE 3
            CMP     AL,NUMERIC
            JZ      OK
FEHLER:     MOV     AX,FALSE
            REST_REGS
            RET_LOGICAL AX
            RET
OK:         GET_INT 1               ; x-Koordinate
            PUSH    AX
```

```
            GET_INT 2           ; y-Koordinate
            PUSH    AX
            GET_INT 3           ; Farb-Attribut -> AX
            POP     DX          ; y-Koordinate -> DX
            POP     CX          ; x-Koordinate -> CX
            MOV     AH,12       ; BIOS-Funktion
            INT     10H         ; "Punkt setzen"
            REST_REGS
            MOV     AX,TRUE     ; .T. zurückgeben
            RET_LOGICAL AX
            RET
G_DOT       ENDP
CODESEG     ENDS
            END
```

Diese Funktion setzt einen Punkt auf dem Grafikschirm in der angegebenen Farbe.

Die x-Koordinate muß zwischen 0 und 319 (619 bei Modus 6) liegen. Der erlaubte Bereich für die y-Koordinate liegt zwischen 0 und 199. Der Punkt x = 0, y = 0 liegt dabei auf der linken oberen Bildschirm-Ecke.

Als Farb-Angaben sind Grundwerte von 0 bis 3 zulässig. Farbe 0 erzeugt einen Punkt in der Hintergrundfarbe, d.h. ein an dieser Stelle vorhandener Punkt wird gelöscht. Wenn Sie zu den Grundwerten 128 addieren, wird das Attribut des neu zu setzenden Punktes mit dem an dieser Stelle zuvor definierten Attribut EXKLUSIV-ODER verknüpft. Damit kann man erreichen, daß beim Aufeinandertreffen von Punkten eine andere Farbe erzeugt wird. Das ist besonders interessant, wenn zwei Grafikdarstellungen überlagert werden.

9.8.6 Linie zeichnen

```
; G_LINE.ASM
;
; von Günther Daubach
;
; Syntax:   logVar = G_LINE(<x1>,<y1>,<x2>,<y2>,<Farbe>)
;
; Rückgabe: .T., wenn die angegebenen Parameter korrekt sind,
;           sonst .F.
;
```

```
; Funktion: Zeichnet eine Linie zwischen den Punkten X1,Y1 und
;           x2,y2 auf dem Grafikschirm. Farbe 0 löscht eine
;           Linie, Farbe 1...3 zeichnet die Linie in der ange-
;           gebenen Farbe.
;
;           Bei Farbwerten 128...131 wird das Farb-Attribut
;           der Linienpunkte mit den an dieser Stelle zuvor
;           gespeicherten Farbattributen EXKLUSIV-ODER
;           verknüpft.

            PUBLIC   G_LINE
            INCLUDE  EXTENDA.MAC
CODESEG     SEGMENT  'CODE'
            ASSUME   CS:CODESEG, DS:CODESEG
G_LINE      PROC     FAR
            JMP      SHORT CONT
DELTA_X     DW       ?              ; ABS(X2-X1)
DELTA_Y     DW       ?              ; ABS(Y2-Y1)
YHALBE      LABEL    WORD           ; ABS(Y2-Y1)/2
XHALBE      DW       ?              ; ABS(X2-X1)/2
ZAEHLER     DW       ?
X1          DW       ?              ; erster Punkt,  X
Y1          DW       ?              ;                Y
X2          DW       ?              ; zweiter Punkt, X
Y2          DW       ?              ;                y
FARBE       DW       ?              ; Linienfarbe
CONT:       SAV_REGS
            GET_PCOUNT              ; Parameterzahl = 5 ?
            CMP      AL,5
            JNZ      FEHLER         ; nein, Fehler
            GET_PTYPE 1             ; Parameter-Typen prüfen
            CMP      AL,NUMERIC
            JNZ      FEHLER
            GET_PTYPE 2
            CMP      AL,NUMERIC
            JNZ      FEHLER
            GET_PTYPE 3
            CMP      AL,NUMERIC
            JNZ      FEHLER
            GET_PTYPE 4
            CMP      AL,NUMERIC
            JNZ      FEHLER
```

```
        GET_PTYPE 5
        CMP     AL,NUMERIC
        JZ      OK
FEHLER: MOV     AX,FALSE
        REST_REGS
        RET_LOGICAL AX          ; .F. zurückgeben
        RET
OK:     PUSH    DS              ; Clipper DS sichern
        GET_INT 1               ; X1
        PUSH    AX
        GET_INT 2               ; Y1
        PUSH    AX
        GET_INT 3               ; X2
        PUSH    AX
        GET_INT 4               ; Y2
        PUSH    AX
        GET_INT 5               ; Farbe
        PUSH    CS
        POP     DS              ; DS = CS
        MOV     FARBE,AX        ; Parameter sichern
        POP     AX
        MOV     Y2,AX
        POP     AX
        MOV     X2,AX
        POP     AX
        MOV     Y1,AX
        POP     AX
        MOV     X1,AX
        MOV     AX,Y2           ; Delta-y berechnen und SI
        SUB     AX,Y1           ; je nach Vorzeichen vor-
        MOV     SI,1            ; belegen
        JGE     SPCH_Y
        MOV     SI,-1
        NEG     AX              ; ABS(Delta-y)
SPCH_Y: MOV     DELTA_Y,AX
        MOV     AX,X2           ; Delta-x berechnen und DI
        SUB     AX,X1           ; je nach Vorzeichen vor-
        MOV     DI,1            ; belegen
        JGE     SPCH_X
        MOV     DI,-1
        NEG     AX              ; ABS(Delta-x)
```

```
SPCH_X:     MOV     DELTA_X,AX
            CMP     AX,DELTA_Y      ; Steigung bestimmen
            JL      POSITIV
            CALL    NEGATIV
            JMP     FERTIG
POSITIV:    CALL    STEIGG          ; Steigung > 1
FERTIG:     POP     DS              ; Clipper DS
            REST_REGS
            MOV     AX,TRUE         ; .T. zurückgeben
            RET_LOGICAL AX
            RET
G_LINE      ENDP

NEGATIV     PROC    NEAR            ; Steigung < 1
            MOV     AX,DELTA_X      ; ABS(Delta-x) durch
            SHR     AX,1            ; 2 dividieren und
            MOV     XHALBE,AX       ; speichern
            MOV     CX,X1           ; x1 lesen
            MOV     DX,Y1           ; y1 lesen
            MOV     BX,0            ; BX initialisieren
            MOV     AX,DELTA_X      ; Zähler setzen
PKTNEU:     CALL    PUNKT           ; Punkt setzen
            ADD     CX,SI           ; x erhöhen/verkleinern
            ADD     BX,DELTA_Y      ; ABS(Delta-y) addieren
            CMP     BX,XHALBE       ; halbe x-Strecke ?
            JLE     PKTZ
            SUB     BX,DELTA_X      ; ABS(Delta-x) subtrahieren
            ADD     DX,SI           ; y erhöhen/verkleinern
PKTZ:       DEC     ZAEHLER         ; Linie fertig ?
            JGE     PKTNEU          ; nein
            RET
NEGATIV     ENDP

STEIGG      PROC    NEAR
            MOV     AX,DELTA_Y      ; ABS(Delta-y) lesen
            SHR     AX,1            ; durch 2 dividieren
            MOV     YHALBE,AX       ; speichern
            MOV     CX,X1           ; x1 lesen
            MOV     DX,Y1           ; y1 lesen
            MOV     BX,0            ; BX initialisieren
            MOV     AX,DELTA_Y      ; Zähler setzen
```

```
PKTNEU2:    CALL    PUNKT          ; Punkt setzen
            ADD     DX,SI          ; y erhöhen/verkleinern
            ADD     BX,DELTA_X     ; ABS(Delta-x) addieren
            CMP     BX,YHALBE      ; halbe y-Strecke ?
            JLE     PKTZ2
            SUB     BX,DELTA_Y     ; ABS(Delta-y) subtrahieren
            ADD     CX,DI          ; x erhöhen/verkleinern
PKTZ2:      DEC     ZAEHLER        ; Linie fertig ?
            JGE     PKTNEU2        ; nein
            RET
STEIGG      ENDP

PUNKT       PROC    NEAR           ; Punkt auf dem Grafikschirm
                                   ; setzen
            PUSH    BX             ; Register sichern
            PUSH    CX
            PUSH    DX
            PUSH    AX
            PUSH    SI
            PUSH    DI
            MOV     AX,FARBE
            MOV     AH,12          ; BIOS Interrupt
            INT     10H            ; "Punkt setzen"
            POP     DI             ; Register zurückholen
            POP     SI
            POP     AX
            POP     DX
            POP     CX
            POP     BX
            RET
PUNKT       ENDP
CODESEG     ENDS
            END
```

Diese Funktion zeichnet eine Linie auf dem Grafikschirm zwischen zwei Punkten in der angegebenene Farbe.

Die x-Koordinaten müssen zwischen 0 und 319 (619 bei Modus 6) liegen. Der erlaubte Bereich für die y-Koordinaten liegt zwischen 0 und 199. Der Punkt x=0, y=0 liegt dabei auf der linken oberen Bildschirm-Ecke.

Als Farb-Angaben sind Grundwerte von 0 bis 3 zulässig. Farbe 0 erzeugt eine Linie in der Hintergrundfarbe, d.h. vorhandene Punkte werden gelöscht. Wenn Sie zu den Grundwerten 128 addieren, werden die Attribute der neu zu setzenden Punkte mit den an dieser Stelle zuvor definierten Attributen EXKLUSIV-ODER verknüpft. Damit kann man erreichen, daß beim Überschneiden von Linien eine andere Farbe erzeugt wird. Das ist besonders interessant, wenn zwei Linien überlagert werden.

9.8.7 Beispiele für den Aufruf der Grafik-Funktionen

```
G_MODE(4)                          && Color, 320 * 200
G_CLS()                            && Schirm löschen
G_HGR(1)                           && Hintergrund blau
G_PAL(0)                           && Palette 0

y = 5
DO WHILE Y < 200                   && horiz. Linien zeichnen
  G_LINE(0,y,319,y,1)
  y = y + 5
ENDDO
x = 5
DO WHILE x < 320                   && vertik. Linien zeichnen
  G_LINE(x,0,x,119,3)
  x = x + 5
ENDDO
FOR i = 0 to 10                    && Palette wechseln
  G_PAL(i)
  INKEY(1)
NEXT
FOR i = 1 TO 15                    && Hintergrund-Farbe wechseln
  G_HGR(i)
  INKEY(1)
NEXT
G_MODE(3)                          && Text, 80*25, Farbe
RETURN
```

Dieses Clipper-Programm zeichnet ein Raster aus horizontalen und vertikalen Linien auf dem Bildschirm und demonstriert danach die Umschaltung zwischen beiden Farbpaletten und den verschiedenen Hintergrundfarben.

```
FUNCTION g_box
PARAMETERS x1,y1,x2,y2,f
  G_LINE(x1,y1,x2,y1,f)
  G_LINE(x2,y1,x2,y2,f)
  G_LINE(x1,y2,x2,y2,f)
  G_LINE(x1,y1,x1,y2,f)
RETURN .T.
```

Diese Clipper-UDF dient zum Zeichnen von Rahmen im Grafik-Modus. Die fünf Parameter geben die Koordinaten des oberen linken und des unteren rechten Eckpunktes sowie die Linienfarbe an.

```
FUNCTION g_boxf
PARAMETERS x1,y1,x2,y2,f,ff,i,h
PRIVATE z
  G_LINE(x1,y1,x2,y1,f)
  G_LINE(x2,y1,x2,y2,f)
  G_LINE(x1,y2,x2,y2,f)
  G_LINE(x1,y1,x1,y2,f)
  IF h
    z = y1 + i
    DO WHILE z <= y2-i
      G_LINE(x1+1,z,x2-1,z,ff)
      z = z + 1
    ENDDO
  ELSE
    z = x1 + i
    DO WHILE z <= x2-i
      G_LINE(z,y1+1,z,y2-1,ff)
      z = z + i
    ENDDO
  ENDIF
RETURN .T.
```

Diese Cipper-UDF zeichnet einen gefüllten Rahmen auf dem Grafikschirm. Die ersten fünf Parameter entsprechen denen der UDF `G_BOX()`.

Diese Funktion bietet jedoch zusätzlich die Möglichkeit, den Rahmen mit Linien zu füllen. Die Farbe dieser Linien wird mit dem sechsten Parameter `ff` angegeben. Der siebte Parameter `i` gibt an, in welchem Abstand diese Linien gezeichnet werden sollen. Mit dem Wert 1 wird ein völliges Füllen des Rahmens erreicht. Als letzter Parameter muß ein logischer Wert angegeben werden. Bei

`.T.` wird der Rahmen mit horizontalen Linien gefüllt; bei `.F.` werden senkrechte Linien gezeichnet.

Mit diesen Möglichkeiten können z.B. Balkendiagramme mit verschiedenfarbigen und schraffierten Balken dargestellt werden.

Besonders bei großen Rahmen erkennt man, daß die Zeichengeschwindigkeit nicht sehr hoch ist. Bessere Ergebnisse lassen sich hier natürlich mit Assembler-Routinen erreichen.

9.8.8 Grafikschirm in einer Datei speichern

```
; G_WSCR.ASM
;
; von Günther Daubach
;
; SYNTAX:     logVar = G_WSCR(<Dateiname>)
;
; RÜCKGABE:   .T., wenn die angegebenen Parameter korrekt sind
;             und wenn die Datei ohne Fehler geschrieben werden
;             konnte, sonst .F.
;
; FUNKTION:   Speichert den aktuellen Inhalt des Grafikschirms
;             in die angegebene Datei.
;
            PUBLIC G_WSCR
            INCLUDE EXTENDA.MAC
CODESEG     SEGMENT 'CODE'
            ASSUME CS:CODESEG
G_WSCR      PROC    FAR
            JMP     SHORT CONT
HANDLE      DW      ?
CONT:       SAV_REGS
            GET_PCOUNT              ; Parameterzahl = 1 ?
            DEC     AX
            JNZ     FEHLER          ; nein
            GET_PTYPE 1             ; Parameter-Typ character ?
            DEC     AX
            JZ      OK              ; ja
```

```
FEHLER:     MOV     AX,FALSE        ; .F. bei Fehler zurückgeben
            REST_REGS
            RET_LOGICAL AX
            RET
OK:         GET_CHAR 1              ; Zeiger auf Dateinamen
            PUSH    DS              ; Clipper DS retten
            MOV     DS,AX           ; Dateiname - Segment
            MOV     DX,BX           ; Dateiname - Offset
            MOV     CX,0            ; DOS-Funktion
            MOV     AH,3CH          ; Datei anlegen und öffnen
            INT     21H
            JNC     OK1             ; CY = 0 -> kein Fehler
            POP     DS
            JMP     FEHLER
OK1:        PUSH    CS
            POP     DS              ; DS = CS
            MOV     HANDLE,AX       ; Dateihandle sichern
            MOV     BX,AX           ; Handle -> BX
            MOV     AX,0B800H       ; Schirm Segment
            MOV     DS,AX           ; -> DS
            MOV     DX,0            ; Schirm Offset
            MOV     CX,4000H        ; Byte-Zahl
            MOV     AH,40H          ; DOS-Funktion
            INT     21H             ; Datei schreiben
            JNC     OK2             ; Fehler ?
            POP     DS
            JMP     FEHLER
OK2:        MOV     BX,HANDLE       ; Handle lesen
            MOV     AH,3EH          ; DOS-Funktion
            INT     21H             ; Datei schließen
            POP     DS              ; Clipper DS zurück
            JC      FEHLER          ; Fehler beim Schließen ?
            MOV     AX,TRUE         ; .T. zurückgeben
            REST_REGS
            RET_LOGICAL AX
            RET
G_WSCR      ENDP
CODESEG     ENDS
            END
```

Diese Funktion speichert den aktuellen Inhalt des Grafikschirms in eine Datei, deren Name beim Aufruf angegeben werden muß. Wenn die Funktion erfolgreich ausgeführt wurde, gibt sie `.T.` zurück. Bei Fehlern oder bei einem unzulässigen Aufruf-Parameter wird `.F.` zurückgegeben.

9.8.9 Datei in den Grafikschirm lesen

```
; G_RSCR.ASM
;
; von Günther Daubach
;
; SYNTAX:     logVar = G_RSCR(<Dateiname>)
;
; RÜCKGABE:   .T., wenn die angegebenen Parameter korrekt sind
;             und wenn die Datei ohne Fehler gelesen werden
;             konnte, sonst .F.
;
; FUNKTION:   Lädt den Inhalt der angegebenen Datei in den
;             Grafikschirm.
;
            PUBLIC G_RSCR
            INCLUDE EXTENDA.MAC
CODESEG     SEGMENT 'CODE'
            ASSUME CS:CODESEG
G_RSCR      PROC   FAR
            JMP    SHORT CONT
HANDLE      DW     ?
CONT:       SAV_REGS
            GET_PCOUNT            ; Parameterzahl = 1 ?
            DEC    AX
            JNZ    FEHLER         ; nein
            GET_PTYPE 1           ; Parameter-Typ character ?
            DEC    AX
            JZ     OK             ; ja
FEHLER:     MOV    AX,FALSE       ; .F. bei Fehler zurückgeben
            REST_REGS
            RET_LOGICAL AX
            RET
OK:         GET_CHAR 1            ; Zeiger auf Dateinamen
            PUSH   DS             ; Clipper DS retten
            MOV    DS,AX          ; Dateiname - Segment
```

```
            MOV     DX,BX           ; Dateiname - Offset
            MOV     AL,0            ; DOS-Funktion
            MOV     AH,3DH          ; Datei zum Lesen öffnen
            INT     21H
            JNC     OK1             ; CY = 0 -> kein Fehler
            POP     DS
            JMP     FEHLER
OK1:        PUSH    CS
            POP     DS              ; DS = CS
            MOV     HANDLE,AX       ; Dateihandle sichern
            MOV     BX,AX           ; Handle -> BX
            MOV     AX,0B800H       ; Schirm Segment
            MOV     DS,AX           ; -> DS
            MOV     DX,0            ; Schirm Offset
            MOV     CX,4000H        ; Byte-Zahl
            MOV     AH,3FH          ; DOS-Funktion
            INT     21H             ; Datei lesen
            JNC     OK2             ; Fehler ?
            POP     DS
            JMP     FEHLER
OK2:        MOV     BX,HANDLE       ; Handle lesen
            MOV     AH,3EH          ; DOS-Funktion
            INT     21H             ; Datei schließen
            POP     DS              ; Clipper DS zurück
            JC      FEHLER          ; Fehler beim Schließen ?
            MOV     AX,TRUE         ; .T. zurückgeben
            REST_REGS
            RET_LOGICAL AX
            RET
G_RSCR      ENDP
CODESEG     ENDS
            END
```

Diese Funktion ist das "Gegenstück" zu der vorstehenden Schreibfunktion. Sie liest die Datei, deren Namen beim Aufruf angegeben wird und überträgt ihren Inhalt in den Grafikschirm.

Die Funktion gibt `.T.` zurück, wenn sie erfolgreich ausgeführt werden konnte. Bei unzulässigem Aufrufparameter oder wenn die Datei nicht gefunden wurde, wird `.F.` zurückgegeben.

9.8.10 Abschließende Bemerkungen zu den Grafikroutinen

Die hier gezeigten Grafikroutinen sollen in erster Linie einen Eindruck von den Möglichkeiten geben, die Assembler-UDFs für Clipper bieten. Sie sind nicht immer optimal in der Ausführungsgeschwindigkeit und zu einem vollständigen Grafikpaket fehlen sicherlich noch etliche Routinen (z.B. Zeichnen von Kreisen und Schraffuren). Beachten Sie außerdem, daß auf dem Bildschirm keine Texte durch Clipper-Anweisungen erzeugt werden können, wenn ein Grafik-Modus aktiv ist. Zur Text-Ausgabe im Grafik-Schirm sind besondere Routinen nötig.

Interessant sind natürlich besonders für Clipper auch solche Routinen, die in Datenbanken als numerische Werte gespeicherte Informationen in grafische Darstellungen, wie Balken-, Torten- oder Liniendiagramme umwandeln.

Derartige Routinen finden Sie in der von der KRS Unternehmensberatung-EDV vertriebenen Grafik-Toolbox für den Clipper-Compiler. Ein besonderer Vorteil dieses Grafikpaketes ist zweifellos die Möglichkeit, daß auch im Grafik-Modus alle gewohnten Bildschirm Ein- und Ausgabebefehle verwendet werden können. Diese Toolbox wird mit Treibern für unterschiedliche Grafik-Adapter (CGA, Hercules, EGA und VGA) ausgeliefert. Die Treiber werden automatisch beim Start eines Programms geladen, so daß für unterschiedliche Bildschirm-Adapter das Programm nicht neu compiliert oder gelinkt werden muß. Auch die Umrechnung der jeweils unterschiedlichen Hardware-Koordinaten in das system-interne "Weltkoordinatensystem" erfolgt automatisch.

Interessant bei dieser Toolbox ist, daß nur die Basis-Routinen in Assemblersprache geschrieben wurden. Alle höheren Funktionen sind in "reinem" Clipper-Code geschrieben und werden im Quellcode mitgeliefert. Dadurch kann der Anwender die Routinen beliebig ändern oder ergänzen.

10 Anwender-definierte Funktionen (UDFs) in "C"

Die Programmiersprache "C" ist die "Muttersprache" von Clipper, d.h. der größte Teil des Compilers und der Clipper-Library wurden in "C" geschrieben. Es ist daher relativ einfach, eigene C-Funktionen in eine Clipper-compilierte Anwendung einzubauen und aufzurufen.

Die Sprache "C" ist als "Mittelweg" zwischen einer Hochsprache und Assemblersprache gut geeignet, um Programme zu erstellen, deren Quellcode relativ kompakt ist, die aber in der Ausführungsgeschwindigkeit oft recht nahe an die von Assemblerprogrammen herankommt.

10.1 Welcher C-Compiler "paßt" zu Clipper ?

Clipper selbst wurde mit Microsoft C, Version 5.0 entwickelt. Alle Beispiele in diesem Buch sind für diesen C-Compiler geschrieben und es wird empfohlen, ihn auch für eigene C-UDFs zu verwenden. Damit ist sichergestellt, daß die Übergabe von Parametern und die Rückgabe von Werten korrekt abläuft und daß das Speichermodell und das Format der erzeugten .OBJ-Dateien zu Clipper compatibel ist.

Wichtig ist, daß alle C-Module als sogenanntes "großes Speichermodell" (large model) compiliert werden müssen. Zusätzlich müssen einige andere Compiler-Optionen richtig gesetzt werden, damit die Module in Clipper-Applikationen integrierbar sind.

10.2 Welche Compiler-Optionen müssen gesetzt werden ?

Für die aktuelle Clipper-Version Sommer ´87 muß der C-Compiler mit den Optionen

```
/c /AL /ZL /Oalt /FPa /Gs
```

aufgerufen werden, d.h. der Compiler-Aufruf muß nach folgendem Muster durchgeführt werden:

```
CL /c /AL /ZL /Oalt /FPa /Gs <Dateiname>.C
```

10.3 Funktionen aus der MS-C Library

Mit dem C-Compiler liefert Microsoft C-Libraries mit einer Vielzahl von Funktionen, die bei der Entwicklung eines eigenständigen C-Programms voll genutzt werden können.

Der Clipper-Compiler wird mit einer eigenen Library (`CLIPPER.LIB`) geliefert, in der viele dieser Funktionen bereits enthalten sind. Insbesonders wird die Laufzeitsteuerung und -umgebung eines Programms bereits aus dieser Library von Clipper angelegt, so daß diese nicht aus der C-Library übernommen werden darf.

Das bedeutet, daß die C-Quellprogramme auf **keinen** Fall eine `main()`-Funktion enthalten dürfen, sondern nur "normale" Funktionen. Nach Möglichkeit sollten in den C-UDFs keine Library-Funktionen aufgerufen werden, die Ein- und Ausgaben (Bildschirm, Drucker und Dateien) bewirken, wie z.B. `printf()`, da dies Konflikte mit Clipper-Funktionen bringen kann.

In einer Clipper-Anwendung wird man C-Funktionen allgemein für interne Verarbeitungszwecke einsetzen und die Ein-/Ausgabe mit den vorhandenen Clipper-Anweisungen durchführen, die schnell genug sind, so daß dies keine sonderliche Einschränkung bedeutet.

Wenn mit dem Linker Clipper `.OBJ`-Module zusammen mit C `.OBJ`-Modulen gebunden werden, sollte man zunächst als Library lediglich `CLIPPER.LIB` angeben. Meldet der Linker nicht erfüllte Referenzen, die aus den C-Modulen stammen, dann sollte als zusätzliche Library die C-Library angegeben werden. Wichtig ist dabei, daß die C-Library für das **large** Modell verwendet wird. Außerdem **müssen** die Clipper-Library und evtl. andere, eigene Libraries **vor** der C-Library angegeben werden, damit der Linker zunächst versucht, Referenzen aus der Clipper-Library zu erfüllen und erst abschließend die C-Library durchsucht.

10.4 Standard Header-Datei EXTEND.H

Ähnlich wie bei Assembler-UDFs Standard-Vereinbarungen in einer `INCLUDE`-Datei (`EXTENDA.MAC`) definiert wurden, um Assemblerprogramme übersichtlicher und einfacher zu machen, sollte auch für C-UDFs eine Standard Header-Datei mit entsprechenden Definitionen verwendet werden. Zusammen mit dem Clipper Compiler wird die Datei `EXTEND.H` geliefert, die auch von den Beispielen in diesem Buch verwendet wird.

10.5 Parameter-Übergabe und Wert-Rückgabe

Wie bei Assembler-UDFs sollten auch bei C-UDFs *nur* die in Clipper "eingebauten" Standard-Funktionen zur Übernahme von Parametern und zur Rückgabe von Werten verwendet werden, um nicht die internen Abläufe von Clipper zu stören und um zu neuen Clipper-Versionen kompatibel zu bleiben.

Bei C-Funktionen werden dieselben Übernahme- und Rückgabefunktionen verwendet, wie bei Assembler, lediglich ihr Aufruf erfolgt unter C natürlich anders, nämlich einfach als C-Funktion.

In der oben erwähnten Header-Datei `extend.h` sind diese Funktionen als extern deklariert. Weiterhin enthält sie einige Makros, die die Prüfung auf Parameter-Anzahl und Typ vereinfachen.

Hier einige Regeln für den prinzipiellen Aufbau des Quellprogramms für eine C-UDF:

1. Datei `extend.h` einlesen (`#include "extend.h"`)
2. UDF-Namen als Funktion Typ `CLIPPER` deklarieren und *keine* Eingabe-Parameter angeben. (Der Typ `CLIPPER` ist in `extend.h` definiert.)
3. Im Funktionsrumpf ggf. zunächst Parameter-Anzahl und -Typen prüfen und bei Fehlern auf einen Programmteil zur Rückgabe eines Fehlercodes verzweigen.
4. Anwendungs-abhängige Anweisungen
5. Rückgabe eines Wertes mit einer der Standard-Rückgabefunktionen oder Aufruf von `_ret()` wenn kein Wert zurückgegeben werden muß.

Wie immer sagt ein Beispiel mehr als viele Erklärungen:

```
/***
*   wild.c
*
*   von Günther Daubach
*
*   Aufruf:
*   logVar = wild(<Maske>,<Text>)
*
*   Funktion:
*   Zwei Zeichenketten von links nach rechts zeichenweise
*   miteinander vergleichen, wobei die erste davon
*   als Maske "Jokerzeichen" enthalten kann. Dabei
```

```
*   gilt folgendes:
*
*   Zeichen "?": an der entsprechenden Position kann in
*                <Text> jedes beliebige Zeichen stehen.
*   Zeichen "*": ab der entsprechenden Position können in
*                <Text> beliebige Zeichen stehen; der
*                Vergleich wird abgebrochen.
*
*   Rückgabe:
*                .T., wenn Übereinstimmung
*                .F., wenn Abweichung oder wenn unzuläs-
*                     sige Parameter übergeben wurden.
*
*   Anmerkung:   Führende oder folgende Leerzeichen werden
*                mit berücksichtigt.
*
***/

#include "extend.h"

CLIPPER wild()

{
    char *maske;
    char *text;

    if (PCOUNT != 2)            /* Parameter-Anzahl = 2 ? */
    {
         _retl(0);
    }
    else
         if (ISCHAR(1) & ISCHAR(2))      /* beide Parameter
                                            Typ char ? */
         {
              maske = _parc(1);          /* Maske einlesen */
              text  = _parc(2);          /* Text einlesen */

              /* maske und text solange zeichenweise ver-
                 gleichen, wie Zeichen gleich sind oder das
                 Maskenzeichen gleich '?' ist und solange
                 weder in maske noch in text das Endebyte
                 (\0) erreicht ist. */
```

```
            while(((*maske == *text) || (*maske == '?')) &&
                    *text && *maske)
            {
                maske++;   /* Zeiger inkrementieren */
                text++;
            }

            /* Funktion muß .T. liefern, wenn entweder in
               der Maske '*' erreicht ist oder wenn Maske
               und Text bis zum Ende ohne Abweichung durch-
               laufen wurden. */

            _retl((*maske == '*') || (*maske == *text));
        else
        {
            _retl(0);          /* Abweichung, also .F. zurück-
                                  geben */
        }
}
```

Dieses Beispiel demonstriert, wie eine C-UDF die Anzahl und Typen der übergebenen Parameter überprüft und bei Fehlern reagiert.

Die Funktion kann in vielen Clipper-Anwendungen sinnvoll eingesetzt werden, wenn es darum geht, zu prüfen, ob Teile einer Zeichenkette mit einer anderen Zeichenkette übereinstimmen.

Die in der Funktion verwendeten "Jokerzeichen" entsprechen denen, die bei DOS in Dateinamen angegeben werden können.

Beispiele für den Funktionsaufruf:

`wild("abc","abc")`	ergibt `.T.`	
`wild("abc","acc")`	ergibt `.F.`	
`wild("abc","abcd")`	ergibt `.F.`	
`wild("a?c","abc")`	ergibt `.T.`	
`wild("abc","a?c")`	ergibt `.F.`	(Maske und Text können nicht vertauscht werden)
`wild("abc*","abcdefgh")`	ergibt `.T.`	
`wild("abc*","accdefgh")`	ergibt `.F.`	

10.6 Suchen, ob eine bestimmte Gruppe von Zeichen an beliebiger Stelle einer Zeichenkette vorkommt

```
/***
*   lookup.c
*
*   von Günther Daubach
*
*   Aufruf:
*   logVar = lookup(<Auswahl>,<Text>)
*
*   Funktion:
*   Die Zeichenkette <Text> wird untersucht, ob alle in
*   <Auswahl> angegebenen Zeichen an beliebiger Stelle in
*   <Text> enthalten sind.
*
*   Rückgabe:
*                .T., wenn alle Zeichen enthalten sind
*                .F., wenn nicht alle Zeichen enthalten sind
*                     oder wenn unzulässige Parameter
*                     übergeben wurden.
*
***/

#include "extend.h"

CLIPPER lookup()

{
    char *auswahl;
    char *text;
    int  ok;

    if (PCOUNT != 2) {          /* Parameter-Anzahl prüfen */
         _retl(0);
    }
    else
         if (ISCHAR(1) & ISCHAR(2)) {    /* Par.-Typen ok ? */
              auswahl = _parc(1);   /* Auswahlzeichen lesen */
              ok = 1;               /* ok zum Start = .T. */
```

```
            /* Solange nicht das Ende von <Auswahl> erreicht
               ist und noch keine Zeichen fehlt, weiter
               prüfen */

            while (*auswahl && ok) {
                  text = _parc(2); /* Text einlesen */
                  ok = 0;          /* ok vorab auf .F. */

                  /* Nach einem Zeichen von <Auswahl> so-
                     lange in <Text> suchen, bis das Ende
                     von <Text> erreicht ist. */

                  while (*text) {
                        /* Wenn Zeichen gefunden wurde,
                           ok = .T. setzen */

                        if (*text == *auswahl) {
                              ok = 1;
                        }
                        text++;     /* nächstes Zeichen in
                                       <Text> */
                  }
                  auswahl++;        /* nächstes Zeichen von
                                       <Auswahl> */
            }
            _retl(ok);              /* Merker zurückgeben */
      }
      else {
            _retl(0);               /* Bei ungültigen Para-
                                       meter-Typen .F. zu-
                                       rückgeben */
      }
}
```

Diese Funktion ist besonders dann hilfreich, wenn z.B. in einer Datenbank ein Feld enthalten ist, in dem Kennbuchstaben ohne festgelegte Reihenfolge stehen und man überprüfen will, ob bestimmte Buchstaben darin enthalten sind.

Beispiele für den Funktionsaufruf:

`lookup("a","abc")`	ergibt `.T.`	
`lookup("a","cab")`	ergibt `.T.`	
`lookup("d","abc")`	ergibt `.F.`	
`lookup("ac","bac)`	ergibt `.T.`	
`lookup("abc","cba")`	ergibt `.T.`	
`lookup("abc","cbef")`	ergibt `.F.`	
`lookup("A")`	ergibt `.F.`	(falsche Parameterzahl)
`lookup(1,2)`	ergibt `.F.`	(falsche Parametertypen)

10.7 Zählen, wieviele Zeichen einer Gruppe an beliebiger Stelle in einer Zeichenkette vorkommen

```
/***
*   nchars.c
*
*   von Günther Daubach
*
*   Aufruf:
*   numVar = nchars(<Auswahl>,<Text>)
*
*   Funktion:
*   Die Zeichenkette <Text> wird darauf untersucht, wieviele
*   der in <Auswahl> angegebenen Zeichen an beliebiger Stelle
*   in <Text> enthalten sind.
*
*   Rückgabe:
*                n    Anzahl der Zeichen (0, wenn kein Zeichen
*                     enthalten ist)
*               -1    wenn unzulässige Parameter übergeben
*                     wurden.
*
***/

#include "extend.h"

CLIPPER nchars()
```

```
{
    char *auswahl;
    char *text;
    int  zaehler;

    if (PCOUNT != 2) {          /* Parameter-Anzahl prüfen */
         _retni(-1);
    }
    else
         if (ISCHAR(1) & ISCHAR(2)) {    /* Par.-Typen ok ? */
              auswahl = _parc(1);   /* Auswahlzeichen lesen */
              zaehler = 0;          /* Variablen müssen ini-
                                       tialisiert werden, um
                                       Überraschungen zu ver-
                                       meiden ! */

              /* Solange nicht das Ende von <Auswahl> erreicht
                 ist, weiter prüfen */

              while (*auswahl) {
                   text = _parc(2); /* Text einlesen */

                   /* Nach einem Zeichen von <Auswahl> so-
                      lange in <Text> suchen, bis das Ende
                      von <Text> erreicht ist. */

                   while (*text) {
                        /* Wenn Zeichen gefunden wurde,
                           zaehler erhöhen */

                        if (*text == *auswahl) {
                             zaehler++;
                        }
                        text++;     /* nächstes Zeichen in
                                       <Text> */
                   }
                   auswahl++;       /* nächstes Zeichen von
                                       <Auswahl> */
              }
              _retni(zaehler);      /* Zähler zurückgeben */
         }
```

```
        else {
             _retni(-1);              /* Bei ungültigen Para-
                                         meter-Typen -1 zu-
                                         rückgeben */
        }
}
```

Diese Funktion ähnelt der vorangehend beschriebenen Funktion `lookup()`. Hier wird jedoch gezählt, wie oft die in `<Auswahl>` enthaltenen Zeichen insgesamt in `<Text>` vorkommen.

Wenn Sie in `<Auswahl>` nur ein Zeichen angeben, können Sie natürlich die Funktion auch dazu verwenden, um festzustellen, wie oft ein bestimmtes Zeichen in `<Text>` vorkommt.

Hier wird der Fall, daß fehlerhafte Parameter übergeben werden, so behandelt, daß der Wert -1 zurückgegeben wird, er kann also im Programm erkannt werden.

10.8 Abfragen von Clipper-SET Parametern

Die nachfolgenden C-Funktionen sind im Buch in einem Quellprogramm zusammengefaßt. Auf der beiliegenden Diskette sind sie in einzelne Dateien (`CL01.C` bis `CL24.C`) aufgeteilt. Dadurch wird erreicht, daß die erzeugten Objekt-Module (`CL01.OBJ` bis `CL24.OBJ`) einzeln in die Bibliothek-Datei `TT87.LIB` aufgenommen werden; damit wird beim Einbinden in eine Clipper-Anwenung nur der tatsächlich erforderliche Code übernommen.

```
/****
*   Aufrufe:                    Rückgaben:
*   logVar  = is_altern()       .T., wenn SET ALTERNATE ON
*   logVar  = is_altc()         .T., wenn SETCANCEL(.T.)
*   logVar  = is_bell()         .T., wenn SET BELL ON
*   logVar  = is_century()      .T., wenn SET CENTURY ON
*   logVar  = is_conf()         .T., wenn SET CONFIRM ON
*   logVar  = is_cons()         .T., wenn SET CONSOLE ON
*   logVar  = is_curs()         .T., wenn SET CURSOR ON
*   logVar  = is_dele()         .T., wenn SET DELETED ON
*   logVar  = is_delim()        .T., wenn SET DELIMITERS ON
*   logVar  = is_devpr()        .T., wenn SET DEVICE TO PRINT
*   logVar  = is_esc()          .T., wenn SET ESCAPE ON
```

```
*   logVar  = is_exact()        .T., wenn SET EXACT ON
*   logVar  = is_exclus()       .T., wenn SET EXCLUSIVE ON
*   logVar  = is_fixed()        .T., wenn SET FIXED ON
*   logVar  = is_intens()       .T., wenn SET INTENSITY ON
*   logVar  = is_print()        .T., wenn SET PRINT ON
*   logVar  = is_score()        .T., wenn SET SCOREBOARD ON
*   logVar  = is_soft()         .T., wenn SET SOFTSEEK ON
*   logVar  = is_wrap()         .T., wenn SET WRAP ON
*   logVar  = is_uniq()         .T., wenn SET UNIQUE ON
*   numVar  = dec_set()         Wert von SET DECIMALS TO
*   numVar  = marg_set()        Wert von SET MARGIN TO
*   charVar = defa_set()        Lauferk gem. SET DEFAULT TO
*   charVar = path_set()        Pfad gem. SET PATH TO
*
*   Funktion:
*   Die Funktionen geben Informationen über die mit den SET-
*   Kommandos in Clipper gesetzten Parameter zurück. Dabei
*   werden die internen Merker von Clipper abgefragt.
*
***/

#include "extend.h"

extern int _alternate_on;
CLIPPER is_altern() {
    _retl(_alternate_on ? 1 : 0);
}

extern int _altc_on;
CLIPPER is_altc() {
    _retl(_altc_on ? 1 : 0);
}

extern int _bell_on;
CLIPPER is_bell() {
    _retl(_bell_on ? 1 : 0);
}

extern int _century_on;
CLIPPER is_century() {
    _retl(_century_on ? 1 : 0);
}
```

```
extern int _confirm_on;
CLIPPER is_conf() {
    _retl(_confirm_on ? 1 : 0);
}

extern int _console_on;
CLIPPER is_cons() {
    _retl(_console_on ? 1 : 0);
}

extern int _curs_on;
CLIPPER is_curs() {
    _retl(_curs_on ? 1 : 0);
}

extern int _deleted_on;
CLIPPER is_dele() {
    _retl(_deleted_on ? 1 : 0);
}

extern int _delimiters_on;
CLIPPER is_delim() {
    _retl(_delimiters_on ? 1 : 0);
}

extern int _exact_on;
CLIPPER is_exact() {
    _retl(_exact_on ? 1 : 0);
}

extern int _exclu_on;
CLIPPER is_exclus() {
    _retl(_exclu_on ? 1 : 0);
}

extern int _fixed_on;
CLIPPER is_fixed() {
    _retl(_fixed_on ? 1 : 0);
}
```

```
extern int _intensity_on;
CLIPPER is_intens() {
    _retl(_intensity_on ? 1 : 0);
}

extern int _print_on;
CLIPPER is_print() {
    _retl(_print_on ? 1 : 0);
}

extern int _score_on;
CLIPPER is_score() {
    _retl(_score_on ? 1 : 0);
}

extern int _softseek;
CLIPPER is_soft() {
    _retl(_softseek ? 1 : 0);
}

extern int _escape_on;
CLIPPER is_esc() {
    _retl(_escape_on ? 1 : 0);
}

extern int _device_p;
CLIPPER is_devpr() {
    _retl(_device_p ? 1 : 0);
}

extern int _wrap_on;
CLIPPER is_wrap() {
    _retl(_wrap_on ? 1 : 0);
}

extern int _unique_on;
CLIPPER is_uniq() {
    _retl(_unique_on ? 1 : 0);
}
```

```
extern int _decimals;
CLIPPER dec_set() {
    _retni(_decimals);
}

extern int _margin;
CLIPPER marg_set() {
    _retni(_margin);
}

extern char _default_drive;
CLIPPER defa_set() {
    _retc(&_default_drive);
}

extern char _path;
CLIPPER path_set() {
    _retc(&_path);
}
```

Diese Funktionen gestatten es, bestimmte Werte abzufragen, die mit den Clipper-SET-Anweisungen definiert wurden, bzw. von Clipper mit Standardwerten belegt wurden. Diese UDFs können besonders in anderen universell geschriebenen Clipper-UDFs oder in Prozeduren eingesetzt werden, wenn diese auf bestimmte SET-Parameter unterschiedlich reagieren müssen. So könnte z.B. eine Ausgabeprozedur prüfen, ob eine Ausgabe auf dem Drucker erfolgt (mit `is_print()` und `is_devpr()`) und in diesem Fall mit der Funktion `isprinter()` prüfen, ob der Drucker bereit ist. (`isprinter()` ist eine Funktion, die zusammen mit Clipper in der Zusatz-Bibliothek EXTEND.LIB geliefert wird.)

Die Funktion `is_curs()` kennen Sie bereits als Assembler-UDF. Sie wurde hier aufgenommen, um zu zeigen, daß in vielen Fällen eine C-Routine einfacher und (als Quellcode) kürzer zu realisieren ist.

11 Ein einfaches Testprogramm für Funktionen

Besonders zum Testen selbst geschriebener UDFs, aber auch zum Ausprobieren Clipper-interner Funktionen ist folgendes kleine Programm recht nützlich:

```
*****
* FTEST.PRG
*
* Programm zum Testen von Funktionen
*
*****

*
* ACHTUNG: hier eigene UDFs mit der EXTERN-Anweisung angeben,
*          wenn diese in einer .LIB-Datei enthalten sind.
*          z.B.:

EXTERN wild, isdrive

* Beim Linklauf müssen die entsprechenden Bibliothek-
* dateien mit eingebunden werden

CLEAR SCREEN
fkt = SPACE(50)
@ 10,0 SAY "Funktion:"
@ 12,0 SAY "Ergebnis:"
DO WHILE .T.
  @ 10,12 GET fkt PICTURE "@K"
  READ
  IF UPPER(TRIM(fkt)) = "ENDE"
    EXIT
  ENDIF
  @ 12,12 SAY SPACE(60)
  @ 12,12 SAY &fkt
ENDDO
CLEAR SCREEN
QUIT
```

Das Programm liest einen Text ein, der einen Ausdruck enthalten muß, und gibt anschließend über die Makro-Funktion das Ergebnis des Ausdruck aus. Dieser Vorgang wird solange wiederholt, bis der Text `ende` eingegeben wird.

Innerhalb einer laufenden Clipper-Anwendung kann man, sofern `DEBUG.OBJ` mit eingebunden wurde, auch im Debugger bequem Funktionen testen, indem man den Menüpunkt "`Display Expression`" (Ausdruck anzeigen) anwählt und dort eine Funktion eingibt.

12 Prüfen und Anlegen von Dateien

Bei allen Anwendungsprogrammen, unabhängig davon, ob in Clipper oder in dBASE, ist es wichtig, daß die erforderlichen Dateien in richtiger Form zur Verfügung stehen. Es sollte daher in jedem Programm am Anfang eine Anweisungsfolge "eingebaut" werden, die prüft, ob alle benötigten Dateien vorhanden sind und ggf. fehlende Dateien anlegen.

Die Abfrage, ob eine Datei existiert, kann mit der Funktion `file()` durchgeführt werden, z.B.:

```
...
IF !FILE("KUNDEN.DBF")
  DO ANLEGEN
ENDIF
...
```

Ebenso können auch die Index-Dateien geprüft und ggf. neu angelegt werden:

```
...
USE KUNDEN
IF !FILE("KDNAME.NTX")
  INDEX ON UPPER(NAME) TO KDNAME
ELSE
  SET INDEX TO KDNAME
ENDIF
...
```

Beachten Sie, daß der Dateiname in der `file()`-Funktion vollständig angegeben werden muß (Name und Zusatz).

Das nächste Programmbeispiel verwendet eine Datenbank mit folgender Struktur:

KUNDEN.DBF

Feld	Feldname	Typ	Länge	Dez
1	NAME	Zeichen	25	0
2	VORNAME	Zeichen	25	0
3	PLZ	Numerisch	4	0
4	ORT	Zeichen	25	0
5	STRASSE	Zeichen	25	0
6	SALDO	Numerisch	9	2

Zur Datei gehört eine Indexdatei (KDNAME.NTX), die mit UPPER(name) gebildet wird.

Zu Beginn eines Programms, das diese Dateien verwendet, sollten folgende Anweisungen stehen:

```
IF !FILE("kunden.dbf")                  && Kundendatei vorh. ?
  DO ANLEGEN                            && nein, anlegen
ENDIF
USE kunden                              && Kundendatei öffnen
IF !FILE("kdname.ntx")                  && Indexdatei vorh. ?
  INDEX ON UPPER(name) TO kdname        && nein, anlegen
ELSE
  SET INDEX TO kdname                   && ja, aktivieren
ENDIF
* weitere Anweisungen.....

*
* Datenbank-Struktur anlegen
*
PROCEDURE ANLEGEN
  CREATE temp                           && Strukturdatei erzeugen
  APPEND BLANK                          && Felder definieren
  REPLACE field_name WITH "NAME"
  REPLACE field_type WITH "C"
  REPLACE field_len WITH 25
  REPLACE field_dec WITH 0
  APPEND BLANK
  REPLACE field_name WITH "VORNAME"
  REPLACE field_type WITH "C"
  REPLACE field_len WITH 20
  REPLACE field_dec WITH 0
  APPEND BLANK
  REPLACE field_name WITH "PLZ"
  REPLACE field_type WITH "N"
  REPLACE field_len WITH 4
  REPLACE field_dec WITH 0
  APPEND BLANK
  REPLACE field_name WITH "ORT"
  REPLACE field_type WITH "C"
  REPLACE field_len WITH 25
  REPLACE field_dec WITH 0
```

```
  APPEND BLANK
  REPLACE field_name WITH "STRASSE"
  REPLACE field_type WITH "C"
  REPLACE field_len WITH 30
  REPLACE field_dec WITH 0
  APPEND BLANK
  REPLACE field_name WITH "SALDO"
  REPLACE field_type WITH "N"
  REPLACE field_len WITH 9
  REPLACE field_dec WITH 2
  CREATE kunden FROM temp              && kunden.dbf anlegen
  ERASE temp                           && Strukturdatei löschen
  IF FILE("kdname.ntx")                && evtl. vorhandene alte
    ERASE kdname.ntx                   && Indexdatei löschen
  ENDIF
RETURN
```

In diesem Programm-Abschnitt wird zunächst geprüft, ob die Datei `kunden.dbf` vorhanden ist. Falls das nicht der Fall ist, wird sie mit der Prozedur ANLEGEN neu erzeugt.

Danach öffnet das Programm die Datei `kunden.dbf` und prüft, ob die Index-Datei `kdname.ntx` existiert. Falls nicht, so wird sie neu aufgebaut und dadurch gleichzeitig aktiviert. Wenn die Index-Datei vorhanden ist, wird sie mit der SET INDEX TO-Anweisung aktiviert.

Zum Anlegen der Datei `kunden.dbf` dient die Prozedur ANLEGEN, die exemplarisch zeigt, wie man mit Clipper per Programm Datenbank-Strukturen anlegen kann, ohne daß man das Hilfsprogramm CREATE aktivieren muß.

Zunächst wird mit der Anweisung CREATE `temp` eine Struktur-Datei `temp.dbf` erzeugt, die folgende Struktur hat:

TEMP.DBF

Feld	Feldname	Typ	Länge	Dez
1	FIELD_NAME	Zeichen	10	0
2	FIELD_TYPE	Zeichen	1	0
3	FIELD_LEN	Numerisch	3	0
4	FIELD_DEC	Numerisch	3	0

Diese Struktur ist von Clipper fest vorgegeben. Mit der Anweisung CREATE <Dateiname> FROM <Strukturdatei> kann eine Datenbank erzeugt werden.

Dazu ist es jedoch erforderlich, daß zuvor für jedes Feld in der zu erzeugenden Datenbank in der Strukturdatei ein Satz angefügt wird, in dem die Feld-Informationen eingetragen werden:

Feldname	->	`FIELD_NAME`
Feldtyp	->	`FIELD_TYPE` (`C`, `N`, `L`, `D` oder `M`)
Feldlänge	->	`FIELD_LEN`
Dezimalstellenen	->	`FIELD_DEC`

Beachten Sie, daß die Feldlänge bei den folgenden Feldtypen den jeweils angegebenen Wert haben *muß*:

`L` (logisch)	1
`D` (Datum)	8
`M` (Memo)	10

In der Prozedur `ANLEGEN` wird nach dem Erzeugen der Datenbank die Struktur-Datei `temp.dbf` mit der Anweisung `ERASE temp.dbf` wieder gelöscht.

Falls eine Indexdatei `kdname.ntx` vorhanden ist, wird sie gelöscht, damit im Hauptprogramm anschließend eine neue Indexdatei erzeugt wird, die auf die neue Datenbank abgestimmt ist.

Durch das hier gezeigte Prüfen und automatische Anlegen fehlender Dateien kann eine Anwendung ohne "vorgefertigte" Dateien ausgeliefert werden. Wenn es erforderlich ist, daß bestimmte Datenbanken bereits feste Informationen enthalten, so kann man dies ebenfalls in der Prozedur `ANLEGEN` durchführen.

Besonders wichtig ist auch die Prüfung der Index-Dateien. Sollte einmal der Fall eintreten, daß eine Index-Datei nicht mehr konsistent ist, d.h. ihr Inhalt stimmt nicht mehr mit den Schlüssel-Infirmationen der Datenbank überein, dann kann man einen Neuaufbau der Index-Datei erreichen, indem man sie einfach im Betriebssystem löscht und anschließend wieder das Anwendungsprogramm startet.

Besonders dann, wenn man Datenbanken sowohl mit Clipper als auch mit dBASE III bearbeitet, wird die Konsistenz der Clipper-Indexdateien zerstört, wenn man mit dBASE Inhalte von Schlüsselfeldern verändert.

Der automatische Index-Aufbau ist zudem wesentlich einfacher, als mit dem Programm `INDEX`, das zum Lieferumfang von Clipper gehört, da man hierbei wissen muß, mit welchen Schlüssel-Ausdrücken die Index-Dateien zu bilden sind.

12.1 Bestimmen eines Schlüssel-Ausdrucks einer Index-Datei

Wenn Sie einmal nicht mehr wissen sollten, nach welchem Schlüssel-Ausdruck eine Index-Datei gebildet wurde, dann können Sie diese Information aus der Index-Datei selbst lesen. Mit einem Versatz von 22 Bytes vom Dateianfang steht in jeder Index-Datei der Schlüsselausdruck.

Sie können diese Information mit dem DOS-Kommando DEBUG auslesen (Ihre Eingaben sind fett gedruckt):

```
debug kdname.ntx
-d
1864:0100 06 00 01 00 00 04 00 00-00 00 00 00 21 00 19 00 ............!...
1864:0110 00 00 20 00 10 00 75 70-70 65 72 28 6E 61 6D 65 ......upper(name
1864:0120 29 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 )...............
...
-q
```

Meistens funktioniert auch die "Brutal-Methode":

```
type kdname.ntx
```

In diesem Fall erscheinen auf dem Bildschirm etliche unsinnige Zeichen, aber kurz nach Beginn der Ausgabe sehen Sie im Klartext den Index-Ausdruck.

12.2 Hilfsprogramm zum Generieren der Befehlsfolge zum Anlegen einer Dateistruktur

Wie Sie am obigen Beispielprogramm zum automatischen Anlegen einer Dateistruktur sehen können, müssen Sie einige Schreibarbeit leisten, besonders wenn die anzulegende Datenbank viele Felder hat. Da andererseits sehr oft die Datenbanken während der Entwicklungsphase entweder mit dem CREATE-Hilfsprogramm oder mit dBASE III angelegt werden, liegt es nahe, ein Programm zu schreiben, das eine vorhandenen Datenbankstruktur liest und daraus einen Programmtext zum Anlegen dieser Struktur erzeugt.

```
*
* MAKESTRU.PRG
*
CLEAR SCREEN
STORE SPACE(8) TO dname, pname          && Dateinamen initialis.
@ 10,10 SAY "DBF-Name"                  && und lesen
@ 12,10 SAY "PRG-Name"
@ 10,23 GET dname
@ 12,23 GET pname
READ
dname = TRIM(dname) + ".dbf"            && Zusatz anfügen
pname = TRIM(pname) + ".prg"
IF !FILE("&dname")                      && .DBF-Datei vorhanden ?
  @ 15,10 SAY ".DBF-Datei nicht vorhanden"
  QUIT
ENDIF
USE(dname)                              && .DBF-Datei öffnen
COPY TO temp STRUCTURE EXTENDED         && Struktur kopieren
USE temp                                && Struktur-Datei öffnen
GO TOP

SET PRINTER TO &pname                   && Druckausgabe in .PRG-
*                                       && Datei umleiten
SET PRINTER ON
SET CONSOLE OFF
*
* einleitende Programmzeilen aufbauen
*
?? '*******************************************************'
?  '* Prüfen, ob Datei "' + UPPER(dname) + '" vorhanden ist" ;
 + ' und ggf. anlegen'
?  '*******************************************************'
?
? 'IF !FILE("'  + dname + '")'
? '  CREATE temp"
@ 15,10 say "Feld"
```

```
* für jedes Feld der Datenbank entspechende Anweiungen
* schreiben
DO WHILE !EOF()
  @ 15,23 SAY RECNO() PICT "9999"
  @ 15,28 SAY field_name
  ? '  APPEND BLANK'
  ? '  REPLACE field_name WITH "' + ;
       TRIM(field_name) + '"'
  ? '  REPLACE field_type WITH "' + ;
       field_type + '"'
  ? '  REPLACE field_len WITH ' + ;
       LTRIM(STR(field_len,3,0))
  ? '  REPLACE field_dec WITH ' + ;
       LTRIM(STR(field_dec,3,0))
  SKIP
ENDDO
*
* abschließende Programmzeilen anfügen
*
? '  CREATE ' + dname + ' FROM temp'
? '  ERASE temp.dbf'
? 'ENDIF'
?
? '*********************************************************'
? '* ENDE'
? '*********************************************************'
SET PRINTER OFF
SET CONSOLE ON
SET PRINTER TO

ERSE temp.dbf
CLEAR SCREEN
QUIT
```

Das mit dem obigen Programm erzeugte Programm-Modul können Sie entweder in einen anderen Programmtext hineinkopieren oder als getrenntes Modul mit einer DO-Anweisung aufrufen.

13 Die kontext-abhängige Hilfe - HELP.PRG

Jedes anwender-freundliche Programm sollte mit integrierten Hilfe-Funktionen ausgestattet sein, die es dem Bediener bei der Ausführung eines Programms erlauben, direkt auf dem Bildschirm Hilfetexte anzuzeigen. Wichtig ist dabei, daß der Anwender jeweils zu dem Programmpunkt Hilfe geboten bekommt, den er gerade aktiviert hat.

Clipper bietet (im Gegensatz zu dBASE III) für derartige Hilfe-Systeme eine ausgezeichnete Unterstützung: `HELP.PRG`.

Immer dann, wenn ein Clipper-Programm auf eine Eingabe wartet (außer bei der `INKEY()`-Funktion), bewirkt ein Druck auf die Funktionstaste `F1`, daß eine Prozedur `HELP` aufgerufen wird (falls vorhanden).

An diese Prozedur werden beim Aufruf drei Parameter übergeben:

1. Prozedur — Zeichenvariable, die den Namen der Prozedur enthält, aus der heraus die `F1`-Taste betätigt wurde.
2. Zeile — Programmzeilen-Nummer, in der die Eingabe-Anweisung steht, bei der die `F1`-Taste betätigt wurde.
3. Variable — Name der Variablen, auf die sich die Eingabe-Anweisung bezieht, bei der die `F1`-Taste betätigt wurde.

Besonders der erste und der dritte Parameter erlauben es, innerhalb `HELP` zu erkennen, aus welchem Programmteil Hilfe angefordert wurde und mit entsprechend unterschiedlichen Hilfemeldungen zu reagieren.

Der allgemeine Aufbau von `HELP` sollte nach folgendem Schema erfolgen:

```
PROCEDURE HELP
PARAMETERS proz, zeile, var
PRIVATE puffer
IF proz = "HELP"                    && rekursiven Aufruf sperren
  RETURN
ENDIF

SAVE SCREEN TO puffer               && Bildschirm sichern
```

```
DO CASE                                && Hilfe auswählen
  CASE proz = "Modul_1"
    DO CASE
      CASE var = "Var_1"
        DO Hilfe_1
      CASE var = "Var_2"
        DO Hilfe_2
      * usw.
    ENDCASE
  CASE proz = "Modul_2"
    DO CASE
      CASE var = "Var_3"
        DO Hilfe_3
      CASE var = "Var_4"
        DO Hilfe_4
      * usw.
    ENDCASE
  OTHERWISE
    DO Allg_Hilfe
ENDCASE

* warten auf Taste, z.B.
INKEY(0)
RESTORE SCREEN FROM puffer             && alten Bildschirm zurück
RETURN

PROCEDURE Hilfe_1
*
* Hilfetext zu Modul_1, Var_1 anzeigen
*
RETURN

PROCEDURE Hilfe_2
*
* Hilfetext zu Modul_2, Var_2 anzeigen
*
RETURN

*
* usw.
*
```

```
PROCEDURE Allg_Hilfe
*
* Hilfetext für nicht separat behandelte Fälle anzeigen
*
RETURN
```

Die Prozedur HELP beginnt mit einer PARAMETERS-Anweisung, in der die drei Übergabeparameter angegeben sind.

Wichtig: diese drei Parameter müssen **immer** angegeben werden, selbst dann, wenn sie in HELP nicht ausgewertet werden. Ein Weglassen dieser Zeile bewirkt, daß die Parameter bei jedem Drücken der F1-Taste auf den internen Clipper-Stack geschrieben werden, der dann nach einiger Zeit voll ist, so daß das Programm mit einem Laufzeitfehler anhält. Sie können auf diese Weise ideal eine "Zeitbombe" in ein Programm einbauen.

Ebenso wichtig ist das Sperren eines rekursiven Aufrufes, d.h. es muß verhindert werden, daß innerhalb von HELP ein erneutes Drücken der F1-Taste nochmals die Prozedur HELP aktiviert. Auch das würde zum langsamen Überlaufen des Clipper-Stacks führen.

Da ja auf dem Bildschirm ein Text angezeigt werden soll, aber nicht vorhersehbar ist, in welcher Programmsituation das erfolgt, ist es wichtig, den aktuellen Bildschirm-Inhalt zu sichern, um nach Verlassen von HELP die Original-Anzeige wiederherzustellen. Dazu dient die SAVE SCREEN TO-Anweisung.

Nach diesen einleitenden Maßnahmen schließt sich ein DO CASE...ENDCASE-Block an, in dem zwischen den verschiedenen Prozeduren unterschieden wird, aus denen Hilfe angefordert werden kann. Innerhalb der CASE-Anweisungen sind weitere CASE...ENDCASE-Blöcke enthalten, in denen zwischen den einzelnen Variablen unterschieden wird. Die darin enthaltenen CASE-Anweisungen rufen schließlich die Prozeduren auf, die die jeweils benötigte Hilfe anzeigen.

Da man in der Regel nicht für alle Programmpunkte eine spezifische Hilfe bietet, ist im Haupt-DO CASE...ENDCASE-Block ein OTHERWISE-Zweig enthalten, mit dem ein allgemeiner Text angezeigt wird, z.B.: "keine Hilfe definiert".

Damit der Bediner Zeit hat, den Hilfetext zu lesen, muß nach der Anzeige eine Warteschleife eingebaut werden, die durch Tastendruck verlassen werden kann. Das kann man am einfachsten mit der Funktion INKEY(0) erreichen.

Die abschließende RESTORE SCREEN-Anweisung stellt den alten Bildschirm-Inhalt wieder her.

13.1 Wichtige Punkte, die bei HELP.PRG beachtet werden sollten

13.1.1 Bildschirm löschen

Vor Anzeige eines Hilfetextes wird man in den meisten Fällen den Bildschirm löschen wollen. Wenn im Clipper-Programm GET-Anweisungen enthalten sind, darf das *nicht* mit der CLEAR-Anweisung erfolgen, da CLEAR außer dem Bildschirm auch den gerade aktiven GET...READ-Block abschließt. Verwenden Sie stattdessen die Anweisung `@ 0,0 CLEAR` oder `CLEAR SCREEN`.

13.1.2 GET...READ innerhalb von HELP.PRG

Wenn im Clipper-Programm selbst GET-Anweisungen enthalten sind, dürfen innerhalb von HELP *keine* GET-Anweisungen verwendet werden, wenn nicht sichergestellt ist, daß diese Anweisungen nur dann ausgeführt werden, wenn HELP aus einer anderen Eingabe-Anweisung aufgerufen wurde. Diese Einschränkung ist nötig, da die GET-Anweisung innerhalb von HELP eine passende READ-Anweisung erwartet. Diese READ-Anweisung schließt aber auch die aktiven GET-Anweisungen in der aufrufenden Prozedur ab, so daß dort nach Verlassen von HELP keine Masken-Eingabe mehr möglich ist.

13.1.3 Dateien innerhalb von HELP.PRG

Wenn innerhalb von HELP Dateien angesprochen werden sollen, müssen besondere Vorkehrungen getroffen werden, um nach Verlassen von HELP wieder die Datenbank zu selektieren, die vor dem Aufruf von HELP aktiv war.

Sinnvoll ist es, dabei folgendes Schema einzuhalten:

```
PRIVATE sel_alt

sel_alt = SELECT()
*
* Hier anderen Select-Bereich wählen und Datei-Operationen
* durchführen.
```

```
*
* Soll eine neue Datei geöffnet werden, so verwendet man
* einen Select-Bereich, der im übrigen Programm nicht ver-
* wendet wird.
  SELECT 0                          && nächsten freien SELECT-
*                                      Bereich wählen
* USE info index info1
* ...
* USE
*

SELECT(sel_alt)
```

Wichtig ist, daß zunächst der beim Aufruf aktive Select-Bereich gespeichert wird, bevor ein anderer Select-Bereich angewählt wird. Damit ist es möglich, beim Verlassen von HELP diesen Bereich wieder anzuwählen.

13.2 Universelles HELP-Modul

Wie Sie an obigem Beispiel erkennen, oder bereits aus eigener Erfahrung wissen, ist es recht aufwendig, besonders bei großen Anwendungen, ein umfangreiches Hilfesystem zu programmieren, da mit vielen DO CASE...ENDCASE-Blöcken die verschiedenen Programmsituationen unterschieden werden müssen und zu jedem Punkt eine spezielle Textausgabe zu programmieren ist.

Da nach dem 7. Gesetz von Edsel Murphy und Alfred Klippstein "alle Konstanten variabel" sind, wird sich bald herausstellen, daß die einmal fest programmierten Hilfetexte geändert werden müssen, da ihre Formulierung dem Bediener nicht die Hilfe bietet, die er erwartet.

Das folgende HELP-Modul verwendet eine Datei mit Memo-Feldern, in denen alle Hilfetexte gespeichert sind. Dank der Clipper-Funktion MEMOEDIT() ist es möglich, diese Texte während der laufenden Anwendung neu zu erstellen oder zu editieren. Damit die Hilfetexte schnell gefunden werden, ist die Hilfe-Datei außerdem indiziert.

```
*
* HELP.PRG
*
* Struktur von HILFE.DBF:
*
* Feld  Typ      Länge Dez
* ----  -------  ----- ---
* SCHL  Zeichen  20     0
* INFO  Memo     10     0
*
* Index-Datei HILFE.NTX indiziert auf SCHL
*
*
PARAMETERS proz, zeile, var
PRIVATE buff, sel_alt, c

IF proz = "HELP"                    && rekursiven Aufruf sperren
  RETURN
ENDIF

SAVE SCREEN TO buff                 && Schirm sichern

sel_alt = SELECT()                  && alten Select-Bereich
*                                      sichern

SELECT 0                            && freien Bereich suchen
USE hilfe INDEX hilfe               && Hilfe-Datei öffnen
SEEK proz + var                     && Eintrag suchen

*                                      Rahmen zeichnen
@  8,5,24,75 BOX CHR(201)+CHR(205)+ ;
                 CHR(187)+CHR(186)+ ;
                 CHR(188)+CHR(205)+ ;
                 CHR(200)+CHR(186)+" "
@  8,7 SAY " Hilfe: "
@ 24,7 SAY ;
   " weiter mit beliebiger Taste ... "

IF FOUND()                          && Eintrag gefunden,
*                                      Hilfetext anzeigen
  MEMOEDIT(info,10,7,23,73,.T.,"hlpudf)
  CLEAR TYPEAHEAD                   && Tasteneingabe löschen
```

```
  c = WAITKEY()                          && siehe Clipper-UDFs
  IF CHR(c) = "E"                        && Hilfetext editieren ?
    REPLACE info WITH ;
    MEMOEDIT(info,10,7,23,73,.T.)
  ENDIF
ELSE
*                                        && kein Eintrag gefunden
  @ 10,7 SAY ;
         "Keine Hilfe definiert."
  c = WAITKEY()                          && siehe Clipper-UDFs
  IF CHR(c) = "E"                        && Text eingeben ?
    APPEND BLANK                         && Ja, dann neuen Satz
    REPLACE schl WITH proz + var         && anfügen
    REPLACE info WITH " "
    REPLACE info WITH ;                  && und Text eingeben
      MEMOEDIT(info,10,7,23,73,.T.)

  ENDIF
ENDIF
USE                                      && Hilfedatei schließen
SELECT(sel_alt)                          && Original Select-Bereich
*                                           anwählen
RESTORE SCREEN FROM buff                 && Bildschirm zurückholen
RETURN

FUNCTION hlpudf                          && Funktion zum sofortigen
  KEYBOARD CHR(27)                       && Abbruch von MEMOEDIT()
RETURN 0
```

Diese universelle HELP-Prozedur demonstriert, wie die Hilfetexte in einer Datenbank verwaltet werden können. Wie Sie sehen, wird bei jedem Aufruf der Prozedur die Datei HILFE.DBF neu geöffnet und anschließend wieder geschlossen. Man kann natürlich auch die Datenbank zu Beginn des Hauptprogramms einem Select-Bereich zuordnen und einmal öffnen. Dann entfallen hier die USE-Anweisungen.

Die Eingabe oder Korrektur von Hilfetexten wird im Programm durch die "Geheimtaste" E aktiviert. Wenn Sie diese Taste dem Anwender nicht mitteilen, können Sie das Ändern der Hilfetexte unterbinden, ohne daß Sie für den Fall, daß Sie Änderungen vornehmen möchten ein spezielles Programm starten müssen.

Dieses HELP-Modul können Sie ohne Änderungen in alle von Ihnen erstellten Applikationen einbinden. Sie müssen nur dafür sorgen, daß die Dateien HILFE.DBF und HILFE.NTX vorhanden sind.

Wenn Sie auf einem System im gleichen Verzeichnispfad verschiedene Anwendungen laufen lassen, kommt es zu Konflikten, wenn alle Programme die Hilfe-Dateien unter dem Namen HILFE suchen. In diesem Fall müssen Sie andere Dateinamen verwenden.

Es ist natürlich auch möglich, dieselbe Hilfe-Datei für mehrere Anwendungen zu benutzen. Wenn in mehreren Anwendungen gleiche Prozedur- oder Variablennamen verwendet werden, kann HELP in der Hilfe-Datei jedoch nicht mehr unterscheiden, aus welchem Programm Hilfe angefordert wurde. In diesem Fall sollten Sie in der Datei HELP.DBF das Feld schl z.B. auf 30 Stellen erweitern. Definieren Sie in jeder Anwendung im Hauptprogramm eine PUBLIC-Variable prog und weisen ihr den Programmnamen zu. In HELP.PRG müssen Sie die Zeile

```
REPLACE schl WITH proz + var
```

durch die Zeile

```
REPLACE schl WITH prog + proz + var
```

ersetzen.

Noch umfangreichere Möglichkeiten für ein universelles Hilfesystem bietet Ihnen das zum Clipper Compiler als Zusatz angebotene Programmpaket "KRS-Help". Hier können für jedes Eingabefeld bis zu sieben verschiedene Hilfefenster definiert werden, die auf Wunsch auch automatisch positioniert werden, so daß das aktuelle Eingabefeld in jedem Fall sichtbar bleibt.

13.3 Hilfeaufruf bei INKEY()

Die INKEY()-Funktion eignet sich sehr gut, um bestimmte Programmfunktionsn auszuwählen. Allerdings wird bei einer Eingabe über diese Funktion HELP nicht ktiviert, wenn man die Taste F1 drückt. Das ist prinzipiell auch sinnvoll, da INKEY() grundsätzlich den Code jeder beliebigen Taste, einschließlich Funktionstasten liefern soll. Man kann aber auch bei INKEY() einen Aufruf von HELP simulieren, wie folgender Programmabschnitt zeigt:

```
c = INKEY()
DO CASE
  CASE c = 28                          && F1-Taste
    DO help WITH ;
      PROCNAME(),PROCLINE(),"C"        && Help-Aufruf simulieren
*
* usw.
*
ENDCASE
```

Die `INKEY()`-Funktion liefert bei der Taste `F1` den Code 28. Diese Information kann dazu ausgenutzt werden, um `HELP` als "normale" Prozedur aufzurufen.

14 Die SET KEY ... TO ... Anweisung

Mit der Anweisung `SET KEY <Code> TO <Prozedur>` kann man beliebige Tasten so definieren, daß sie (ähnlich wie `F1` die Prozedur `HELP` aktiviert) Prozeduren aufrufen. Dieser Vorgang läuft asynchron ab, d.h. er kann immer dann auftreten, wenn ein Clipper-Programm auf eine Eingabe wartet (außer bei `INKEY()`); sein Zeitpunkt ist jedoch nicht vorhersehbar, da er vom Bediener bestimmt wird.

Besonders geeignet ist `SET KEY ... TO ...` zum Aufrufen von Programm-Sonderfunktionen. Ein Beispiel dafür finden Sie im Taschenrechnerprogramm, das in Kapitel 19.9 beschrieben ist.

Hier ein Beispiel:

```
SET KEY -9 TO rechner                    && F10 ruft rechner auf
```

Nachdem diese Anweisung durchlaufen wurde, bewirkt ein Drücken der Taste `F10`, daß die Prozedur `rechner` aufgerufen wird.

Für die so aufgerufenen Prozeduren gelten dieselben Regeln, wie für `HELP`. Besonders wichtig ist auch hier, daß an die Prozeduren drei Parameter (`proz`, `zeile` und `var`) übergeben werden, die in der Prozedur in einer `PARAMETERS`-Anweisung *unbedingt angegeben werden müssen*, um zu verhindern, daß bei mehrfachem Aufruf ein Speicherüberlauf auftritt.

Auch die Sperrung eines rekursiven Aufrufes sollte in jedem Fall eingebaut werden, also z.B.:

```
PROCEDURE rechner
PARAMETERS proz, zeile, var
IF proz = "RECHNER"
  RETURN
ENDIF
```

Auch die Aussagen bei `HELP` über das "Retten" des alten Bildschirm-Inhaltes und die Hinweise über `GET`-Anweisungen und das Ansprechen von Dateien haben volle Gültigkeit.

Sie haben es sicher bereits vermutet, `HELP` ist lediglich eine Sonderform von `SET KEY ... TO ...`

Tatsächlich nimmt Clipper in jedem Programm automatisch die Einstellung

```
SET KEY 28 TO help
```

an, sofern keine andere SET KEY 28 TO ... Anweisung im Programm enthalten ist.

Die Zuordnung einer Taste zu einem Prozedur-Aufruf kann durch die Anweisung

```
SET KEY <Code> TO
```

aufgehoben werden. Es gibt bestimmte Fälle, z.B. wenn die MEMOEDIT()-Funktion aktiv ist, in denen man keinen Aufruf von HELP oder von anderen Prozeduren zulassen möchte. Hierzu als Beispiel folgende Programmzeilen:

```
SET KEY 28 TO
MEMOEDIT(text,1,1,24,78,.T.)
SET KEY 28 TO HELP
```

Mit diesen Anweisungen wird die Taste F1 vor dem Aufruf von MEMOEDIT() gesperrt und anschließend wieder zum Aufrufen der Prozedur HELP freigegeben.

Die Anweisung SET KEY ... TO ... kann auch gut dazu verwendet werden, um bestimmte Tasten oder Tastenkombinationen zu ersetzen. So ist es z.B. für den Bediener nicht sehr bequem, die Funktion MEMOEDIT() mit der Tastenkombination Ctrl-W zu beenden. Die folgenden Anweisungen schaffen Abhilfe:

```
SET KEY -1 TO CTRL_W
*
* usw.
*
PROCEDURE CTRL_W
PARAMETERS proz, zeile, var
KEYBOARD CHR(23)
RETURN
```

Durch diese Anweisungen wird die Funktionstaste F2 zur Ctrl-W-Taste "ernannt". Tatsächlich wird beim Drücken von F2 die Prozedur CTRL_W aufgerufen, die mit der KEYBOARD-Anweisung den Code von Ctrl-W in den Tastaturpuffer überträgt, also das Drücken von Ctrl-W simuliert.

Um bei MEMOEDIT() zu bleiben... Das Abbrechen von MEMOEDIT() mit der Esc-Taste ist oft nicht erwünscht, zumal dabei die englische Meldung Abort Edit (Y/N) auf dem Bildschirm erscheint.

Mit den folgenden Anweisungen kann die Esc-Taste "ausgeschaltet" werden

```
SET KEY 27 TO dummy
*
* usw.
*
PROCEDURE dummy
PARAMETERS proz, zeile, var
RETURN
```

Folgenes Programmstück zeigt, wie man eine Funktionstaste zum Abbrechen von MEMOEDIT() definieren kann und wie gleichzeitig eine beliebige Meldung erzeugt wird:

```
*
* usw.
*
a = " "
SET KEY -2 TO abort                    && F3 = Abbruch-Taste
a = MEMOEDIT(a,1,1,15,78,.T.)
SET KEY -2 TO                          && F3 wieder "normal"
*
* usw.
*

PROCEDURE abort
PARAMETERS proz, zeile, var
PRIVATE r, c, ch
r = ROW()                              && alte Cursor-Position
c = COL()                              && speichern
@ 20,10 SAY ;
  "Editor abbrechen ? (J/N)"           && Abbruchmeldung
DO WHILE .T.
  ch = UPPER(CHR(INKEY(0)))            && Taste lesen
  IF ch = "J"                          && Abbruch, dann
    KEYBOARD CHR(27) + "Y"             && Esc + "Y" "schreiben"
    EXIT
```

```
  ELSE
    IF ch = "N"                          && nicht abbrechen
      EXIT
    ENDIF
  ENDIF
ENDDO
@ 20,10 SAY SPACE(24)                    && Meldung löschen
@ r,c SAY ""                             && Cursor auf alte Pos.
RETURN
```

Diese wenigen Beispiele zeigen, wie vielseitig SET KEY ... TO ... eingesetzt werden kann. Ob damit die Bedeutung von Tasten "umgelenkt" wird oder ob z.B. Funktionstasten so belegt werden, daß der nächste oder vorige Satz einer Datenbank gesucht und angezeigt wird...Ihrer Phantasie sind keine Grenzen gesetzt.

15 Die VALID-Klausel bei der GET-Anweisung

Eine der leistungsfähigsten Eigenschaften von Clipper verbirgt sich hinter der zunächst unscheinbaren Möglichkeit, bei einer GET-Anweisung zusätzlich eine VALID-Klausel anzugeben.

Laut Clipper-Handbuch muß hinter der VALID-Klausel ein logischer Ausdruck stehen. Nur wenn dieser Ausdruck den Wert .T. ergibt, wird die Eingabe im betreffenden GET-Feld angenommen und das Feld verlassen.

Im einfachsten Fall kann eine solche GET-Anweisung so aussehen:

```
@ 10,5 GET name VALID !EMPTY(name)
```

das bedeutet, daß in das Feld name mindestens ein Zeichen eingegeben werden muß, da bei einem leeren Feld der Ausdruck !EMPTY(name) den Wert .F. ergibt.

Hier erkennt man, daß im Gegensatz zu dBASE III die GET-Anweisung bei Clipper anders arbeiten muß. Während bei dBASE III die mit GET eingelesenen Werte erst nach Beenden des READ-Blockes bereitstehen, muß Clipper den Wert sofort zur Verfügung stellen, damit er durch die VALID-Klausel geprüft werden kann.

Der logische Ausdruck hinter VALID im obigen Beispiel wurde von einer Funktion geliefert; dabei ist es unerheblich, ob es sich dabei um eine "eingebaute" oder eine anwender-definierte Funktion handelt. Damit ist z.B. auch folgendes möglich:

```
@ 10,5 GET name VALID LEER_P(name)
*
* usw.
*
FUNCTION LEER_P
PARAMETERS nm
IF !EMPTY(nm)
  RETURN .T.
ELSE
  F_MELDUNG(20,"Feld darf " + ;
           "nicht leer bleiben"       && siehe Clipper UDFs
  RETURN .F.
ENDIF
```

Die Funktion LEER_P prüft, ob der übergebene Parameter leer ist, gibt in diesem Fall eine Fehlermeldung aus (die Funktion F_MELDUNG() ist im Kapitel Clipper-UDFs beschrieben) und kehrt mit dem Wert .F. an das aufrufende Programm zurück. Im anderen Fall wird .T. zurückgegeben.

Noch interessanter ist die Verwendung der VALID-Klausel mit einer anwenderdefinierten Funktion, wenn es darum geht, sofort zu prüfen, ob der in ein GET-Feld eingegebene Wert in einer Datenbank enthalten ist, wie folgendes Beispiel zeigt:

```
USE konten INDEX ktnr
@ 10, 0 SAY "Kontonummer"
@ 10,12 GET mktn VALID CHECK(mktn)
*
* weitere GET-Anweisungen
*
READ
*
* usw.
*

FUNCTION CHECK
PARAMETERS nr
SEEK nr
IF FOUND()
  @ 20, 0 SAY bezeich
  @ 20,40 SAY saldo PICTURE "9999999.99"
  RETURN .T.
ELSE
  F_MELDUNG(20,"Konto nicht gespeichert")
  RETURN .F.
ENDIF
```

In diesem Programmbeispiel, das Teil eines Buchhaltungsprogramms sein könnte, wird bei der Erfassung eines Buchungssatzes eine Kontonummer eingegeben (GET mktn VALID CHECK(mktn)). In der VALID-Klausel wird eine Funktion CHECK aufgerufen, die prüft, ob die eingegebene Kontonummer in der Kontendatei gespeichert ist. Wenn das der Fall ist, wird die Bezeichnung des Kontos und der aktuelle Saldo auf dem Bildschirm angezeigt. Im anderen Fall gibt die Funktion eine Fehlermeldung aus. Da die Funktion im Fehlerfall .F. zurückgibt, wird das GET-Feld nicht verlassen, bis entweder eine gespeicherte Kontonummer eingegeben oder bis die Esc-Taste gedrückt wird.

Wer bereits Programmier-Erfahrung mit dBASE III hat, wird die Möglichkeiten besonders zu schätzen wissen, die Clipper zur Plausibilitätsprüfung bei GET-Eingaben mit der VALID-Klausel bietet. Der Versuch, ein ähnliches Verhalten mit den "normalen" Möglichkeiten der GET-Anweisung zu programmieren, führt zu mehrfachen READ-Anweisungen mit umständlich verschachtelten DO...WHILE... ENDDO Blöcken. In Clipper-Programmen kann man dagegen alle zu einer Eingabemaske gehörenden Eingaben mit einer einzigen READ-Anweisung beenden und trotzdem jede einzelne GET-Eingabe sofort auf Gültigkeit prüfen.

Das folgende Beispiel zeigt, wie man mit der VALID-Klausel bereits während der Eingabe Felder in einer Berechnung verknüpfen und das Ergebnis anzeigen kann:

```
CLEAR
STORE 0.00 TO l, b
bez = SPACE(20)
@ 10,1 SAY "Länge:"
@ 12,1 SAY "Breite:"
@ 14,1 SAY "Bezeichnung:"
@ 16,1 SAY "Fläche:"
FLAECHE(l,b)                          && Anfangswert für Fläche
*                                     && ausgeben
@ 10,15 GET l PICTURE "999.99" VALID FLAECHE(l, b)
@ 12,15 GET b PICTURE "999.99" VALID FLAECHE(l, b)
@ 14,15 GET bez
READ
QUIT

FUNCTION FLAECHE                      && Fläche berechnen und
PARAMETERS l, b                       && anzeigen
@ 16,15 SAY l * b PICTURE "999.99"
RETURN .T.
```

In diesem Programmbeispiel werden drei Werte in einem GET...READ-Block eingelesen. Die ersten beiden Werte stellen die Länge und Breite eines Rechtecks dar (z.B. die Seitenlänge eines Teppichs). Sobald Sie im Feld l (Länge) oder b (Breite) eine Eingabe machen, wird sofort die Fläche berechnet und angezeigt. Diesen Vorgang können Sie beliebig oft wiederholen, solange Sie nicht das letzte Feld (Bezeichnung) verlassen.

Auch ein solches Verhalten können Sie mit dBASE III nur durch mehrere READ-Anweisungen und entsprechende Schleifenkonstruktionen erreichen.

Grundsätzlich ist bei Verwendung von UDFs als "Ergebnis-Lieferant" hinter einer VALID-Klausel zu beachten, daß innerhalb der UDF kein GET...READ-Block enthalten ist, da dieses READ auch den übergeordneten GET...READ-Block beenden würde. (Geschachtelte GET...READ-Blöcke stehen immer noch auf meinem persönlichen Wunschzettel an Nantucket.)

Besonders sinnvoll ist die Verwendung von VALID zusammen mit einer UDF auch dann, wenn man die leider noch englischen System-Meldungen von Clipper (z.B. `Invalid Date` und `Range is`) unterdrücken will.

Hierzu finden Sie im Kapitel "Clipper UDFs" die Funktionen `datum_ok()` und `in_range()`. An dieser Stelle sind auch Beispiele für die Verwendung der Funktionen mit VALID gezeigt.

Beachten Sie, daß bei `datum_ok()` keine Datumvariable, sondern eine Zeichenkette im Datum-Format geprüft wird. Dementsprechend muß mit der GET-Anweisung auch eine Zeichenkette und keine Datumsvariable eingelesen werden. Ein "Umbauen" der Funktion, so daß Sie direkt eine Datumsvariable prüft, nützt nichts, da bei einem GET auf eine Datumsvariable die Clipper-interne Datumsprüfung *vor* der Bearbeitung der VALID-Klausel erfolgt und somit die Anzeige "`Invalid date`" nicht unterdrückt wird.

Die Funktion `in_range()` können Sie mit VALID anstelle der RANGE-Klausel einer GET-Anweisung verwenden.

16 Anwendungen mit PROMPT...MENU TO

Fast immer ist es in Anwendungsprogrammen erforderlich, daß der Bediener zwischen mehreren Programmfunktionen wählen kann. Programmteile, die dem Anwender die Funktions-Auswahl ermöglichen, nennt man "Menüs". Bei sehr vielen Programmfunktionen ist es sogar ratsam, aus einem "Hauptmenü" verschiedene "Untermenüs" aufzurufen, aus denen dann die eigentlichen Programmfunktionen aktiviert werden.

Im "konventionellen" dBASE-Stil könnte ein Menüprogramm z.B. so aussehen:

```
wahl = "E"
DO WHILE .T.
  CLEAR
  @  5,10 SAY "Kontendaten - Funktionsauswahl"
  @  6,10 SAY "------------------------------"
  @  8,10 SAY "<A>endern von Konten"
  @  9,10 SAY "<L>öschen von Konten"
  @ 10,10 SAY "<N>eue Konten anlegen"
  @ 12,10 SAY "<E>nde"
  @ 15,10 SAY "Ihre Wahl..." ;
          GET wahl PICTURE "!" VALID wahl $ "ALNE"
  READ

  DO CASE
    CASE wahl = "A"
      DO anlegen
    CASE wahl = "L"
      DO loeschen
    CASE wahl = "N"
      DO neu
    CASE wahl = "E"
      EXIT
  ENDCASE
ENDDO
```

In dieser Programm-Variante werden die Auswahlmöglichkeiten angezeigt und der Bediener muß danach einen Kennbuchstaben eingeben. Beachten Sie, wie durch die Vorbelegung von `wahl` mit "E" dem Bediener eine Auswahl angeboten wird, die er auch mit <Return> übernehmen kann.

Der Aufruf der Programmfunktionen erfolgt dann in einem DO...CASE...ENDCASE-Block.

Viele moderne Programme, wie z.B. Lotus 1-2-3 oder Microsoft Word, bieten zur Funktionsauswahl die Möglichkeit, den Anfangsbuchstaben eines Menüpunktes einzutippen oder mit den Cursortasten eine Markierung auf den gewünschten Punkt zu bewegen und ihn mit <Return> auszuwählen.

Zusätzlich bieten diese Menüs meistens zu jedem Punkt einen kurzen Informationstext, der die Bedeutung eines markierten Menüpunktes erläutert. Besonders durch diesen Informationstext kann man erreichen, daß auf relativ kleiner Bildschirm-Fläche viele Menüpunkte angeordnet werden können.

Das folgende Beispielprogramm zeigt, wie man solche Menüs aufbaut:

```
CLEAR
wahl = 6                                && Auswahl vorbelegen (1
SET MESSAGE TO 24                       && Info in Zeile 24
@ 22, 0 SAY REPLICATE(CHR(205),80)      && Doppel-Linie
DO WHILE .T.
  @ 23, 0 SAY REPLICATE(" ",80)         && Menüzeile löschen  (2
  @ 23, 0 PROMPT "Suchen" MESSAGE ;
          "Suchen eines gespeicherten Kontos"
  @ 23, 7 PROMPT "Löschen" MESSAGE ;
          "Löschen des angezeigten Kontos"
  @ 23,15 PROMPT "Anfügen" MESSAGE ;
          "Anfügen eines neuen Kontos"
  @ 23,23 PROMPT "Drucken" MESSAGE ;
          "Drucken des angezeigten Kontos"
  @ 23,31 PROMPT "Kopieren" MESSAGE ;
          "Kopieren des Kontos in eine Datei"
  @ 23,40 PROMPT "Zurück" MESSAGE ;
          "Funktion beenden und Hauptmenü anzeigen"
  MENU TO wahl

  DO CASE                               && Prozedur-Aufrufe
    CASE wahl = 1
      DO such
*
* weitere CASE-Anweisungen
*
```

```
    CASE WAHL = 6
      EXIT
  ENDCASE
ENDDO
CLEAR
RETURN

PROCEDURE SUCH
  PRIVATE wahl
  @ 23,0 SAY REPLICATE(" ",80)
  wahl = 4                                && (1
  DO WHILE .T.
    @ 23, 0 PROMPT "Bezeichnung" MESSAGE ;
            "Suchen nach einer Konto-Bezeichnung"
    @ 23,12 PROMPT "Konto-Nummer" MESSAGE ;
            "Suchen nach einer gespeicherten Konto-Nummer"
    @ 23,25 PROMPT "Null-Saldo" MESSAGE ;
            "Suchen nach einem Konto mit Saldo = 0"
    @ 23,36 PROMPT "Zurück" MESSAGE ;
            "Funktion beenden und Kontenmenü anzeigen"
  MENU TO wahl
  DO CASE
    *
    * hier weitere CASE-Anweisungen einfügen
    *
    CASE wahl = 4
      EXIT
  ENDCASE
  ENDDO
RETURN
```

Dieser Programm-Abschnitt zeigt, wie man geschachtelte Menüs mit den Clipper-Anweisungen PROMPT...MESSAGE-MENU TO á la 1-2-3 aufbauen kann. Die Arbeitsweise von PROMPT...MENU TO ist sehr ähnlich der von GET...READ. Auch hier bilden die PROMPT-Anweisungen einen Block, der mit der MENU TO-Anweisung abgeschlossen wird. Das Programm verweilt solange in diesem Block, bis der Anwender einen Menüpunkt ausgewählt hat.

Einer der Menüpunkte wird außerdem mit dem Text-Attribut "hervorgehoben" markiert. Diese Markierung kann mit den Pfeiltasten "links", "rechts", "auf" und "ab" auf alle Menüpunkte verschoben werden.

Die Tasten "links" und "auf" bewegen die Markierung auf den Text der vorangehenden PROMPT-Anweisung. Entsprechend bringen die Tasten "rechts" und "ab" die Markierung auf den Text der nächsten PROMPT-Anweisung.

Ein Menüpunkt wird durch Eintippen seines Anfangszeichens angewählt; bei Buchstaben werden Groß- und Kleinbuchstaben gleichwertig behandelt. Man kann auch die Markierung auf den gewünschten Punkt bringen und die Tasten <Return>, <PgUp> oder <PgDn> betätigen.

Nach Auswahl eines Punktes wird der hinter MENU TO angegebenen Variablen ein Zahlenwert zugewiesen, der dem Menüpunkt entspricht. Bei Auswahl des Textes der ersten PROMPT-Anweisung wird der Wert 1, beim zweiten Punkt der Wert 2 usw. zugewiesen.

Als Sonderfall wird der Wert 0 zugewiesen, wenn die <Esc>-Taste gedrückt wurde.

Wird bei einer PROMPT-Anweisung zusätzlich die Option MESSAGE zusammen mit einem Text angegeben, so erscheint dieser Text in einer Bildschirmzeile, wenn der entsprechende Punkt markiert wird. Diese Bildschirmzeile wird mit der Anweisung SET MESSAGE TO <Zeile> definiert. Beachten Sie, daß die SET MESSAGE TO-Anweisung angegeben werden *muß*, wenn Sie die MESSAGE-Option verwenden, da sonst die Texte nicht angezeigt werden.

Die Bildschirm-Zeile, in der diese Texte ausgegeben werden, wird vorher vollständig gelöscht, auch wenn die Texte nicht die gesamte Zeile füllen. Dadurch können unterschiedlich lange Texte ausgegeben werden, ohne daß Teile einer längeren Zeile stehenbleiben, wenn eine kurze Zeile angezeigt wird. Gleichzeitig bedeutet das natürlich auch, daß andere Zeichen (z.B. Rahmenymbole) in dieser Zeile ebenfalls gelöscht werden.

Zu den im obigen Programm in den Kommentaren enthaltenen Ziffern einige Anmerkungen:

1) Die Variable der MENU TO-Anweisung muß vor den PROMPT-Anweisungen mit einem Wert vorbelegt werden. Je nach Vorbelegung wird der entsprechende Menüpunkt markiert. Im Beispielprogramm der Punkt 6 (bzw. 4 in der Prozedur SUCH). Durch diese Vorbelegung kann man einen Menüpunkt vorwählen, der sehr häufig angesprochen wird, so daß der Benutzer zur Auswahl nur die <Return>-Taste drücken muß.

2) Im Gegensatz zur Zeile, in der die MESSAGE-Texte erscheinen, werden die Zeilen nicht gelöscht, in denen PROMPT-Meldungen angezeigt werden. Wenn Sie, wie in obigem Beispiel, mehrere geschachtelte Menüs verwenden, sollten Sie vor der Anzeige neuer Menüpunkte die Anzeige alter Menüpunkte löschen, damit sich keine Texte überlagern.

Wie bereits gesagt, können Menüpunkte durch Eintippen des Anfangszeichens des jeweiligen PROMPT-Textes angewählt werden. Aus diesem Grund sollten alle Texte mit unterschiedlichen Zeichen beginnen. Eine Ausnahme bildet das Leerzeichen. Dieses darf vor einem Text stehen und bleibt unberücksichtigt. Dadurch kann man erreichen, daß die hervorgehobene Markierung eines Menüpunktes breiter ist, als der Text selbst, was oft einen besseren optischen Eindruck gibt. Hier ein Beispiel

```
@ 10,15 PROMPT " Kunden        "
@ 11,15 PROMPT " Lieferanten   "
@ 12,15 PROMPT " Interessenten "
*
* usw.
```

Durch führende und nachgestellte Leerzeichen werden die PROMPT-Texte auf gleiche Länge gebracht, so daß auch die Markierung immer gleich lang ist. Trotzdem können die Punkte durch Eintippen der Buchstaben "K", "L" und "I" ausgewählt werden.

16.1 "Pull Down"-Menüs

Viele moderne Programme verwenden sogenannte "Pull Down" Menüs. Hierbei sind in der oberen Bildschirmzeile die Haupt-Menüpunkte enthalten. Eine Markierung kann mit den Pfeiltasten verschoben werden und unterhalb des markierten Punktes wird ein "Fenster" geöffnet, in dem Unter-Menüpunkte erscheinen, die dann durch Positionieren einer weiteren Markierung ausgewählt werden können.

Mit Clipper läßt sich auch eine solche Menüstruktur aufbauen, wie dieses Beispielprogramm zeigt:

```
CLEAR
flg = " "                          && Merker für Rechts-/
*                                     Linkspfeil
wahl = 1                           && Variablen für die
wahl1 = 1                          && Menü-Auswahl vorbe-
wahl2 = 1                          && legen
wahl3 = 1
wahl4 = 1
rahmen = CHR(201)+CHR(205)+ ;
         CHR(187)+CHR(186)+ ;
         CHR(188)+CHR(205)+ ;
         CHR(200)+CHR(186)         && String f. Doppelrahmen
KEYBOARD CHR(13)                   && ersten Hauptmenüpunkt
*                                     anwählen
DO WHILE .T.
  SET KEY 4 TO                     && Rechts- und Linkspfeil
  SET KEY 19 TO                    && "normal" aktivieren
  @ 1, 0 PROMPT "Essen"            && Hauptmenü
  @ 1, 6 PROMPT "Trinken"
  @ 1,14 PROMPT "Musik"
  @ 1,20 PROMPT "Ende"
  MENU TO wahl                     && Haupt-Auswahl
  SET KEY 4 TO r_esc               && Rechtspfeil "umlenken"
  SET KEY 19 TO l_esc              && Linkspfeil "umlenken"
  DO CASE                          && Haupt-Auswahl
    CASE wahl = 1
      DO essen
    CASE wahl = 2
      DO trinken
    CASE wahl = 3
      DO musik
    CASE wahl = 4
      DO ende
  ENDCASE
ENDDO
CLEAR
RETURN
```

```
PROCEDURE essen                          && erstes Untermenü
  @ 2,0,8,14 BOX rahmen
  @ 3,1 PROMPT " Deutsch      "
  @ 4,1 PROMPT " Italienisch "
  @ 5,1 PROMPT " Chinesisch  "
  @ 6,1 PROMPT " Balkan      "
  @ 7,1 PROMPT " Griechisch  "
  MENU TO wahl1
  DO CASE
    CASE wahl1 = 1
      @ 15,10 SAY "Wiener Schnitzel      "
    CASE wahl1 = 2
      @ 15,10 SAY "Pizza Regina          "
    CASE wahl1 = 3
      @ 15,10 SAY "Nasi Goreng           "
    CASE wahl1 = 4
      @ 15,10 SAY "Agramer Schindelbraten"
    CASE wahl1 = 5
      @ 15,10 SAY "Gyros                 "
  ENDCASE
  IF chk_arrow()                         && Wechsel im Hauptmenü ?
    @ 2,0,8,14 BOX SPACE(9)              && Untermenü-Anzeige
  ENDIF                                  && löschen
RETURN

PROCEDURE trinken                        && zweites Untermenü
  @ 2,6,7,20 BOX rahmen
  @ 3,7 PROMPT " Wein        "
  @ 4,7 PROMPT " Bier        "
  @ 5,7 PROMPT " Alkoholfrei "
  @ 6,7 PROMPT " Spirituosen "
  MENU TO wahl2
  DO CASE
    CASE wahl2 = 1
      @ 16,10 SAY "Steinhauser Frosthelfer"
    CASE wahl2 = 2
      @ 16,10 SAY "Wassersteiner Dünnbräu "
    CASE wahl2 = 3
      @ 16,10 SAY "Kranenberger Urquell   "
    CASE wahl2 = 4
      @ 16,10 SAY "Kracher Qualmrauh      "
  ENDCASE
```

```
  IF chk_arrow()
    @ 2,6,7,20 BOX SPACE(9)
  ENDIF
RETURN

PROCEDURE musik                          && drittes Untermenü
  @ 2,14,7,27 BOX rahmen
  @ 3,15 PROMPT " Beat       "
  @ 4,15 PROMPT " Klassik    "
  @ 5,15 PROMPT " Volksmusik "
  @ 6,15 PROMPT " Schnulze   "
  MENU TO wahl3
  DO CASE
    CASE wahl3 = 1
      @ 17,10 SAY "A hard day's night                    "
    CASE wahl3 = 2
      @ 17,10 SAY "Eine kleine Nachtmusik                "
    CASE wahl3 = 3
      @ 17,10 SAY "Das Bergecho                          "
    CASE wahl3 = 4
      @ 17,10 SAY "Es hängt ein Pferdehalfter an der Wand"
  ENDCASE
  IF chk_arrow()
    @ 2,14,7,27 BOX SPACE(9)
  ENDIF
RETURN

PROCEDURE ende                           && viertes Untermenü
  @ 2,20,4,36 BOX rahmen
  @ 3,21 PROMPT " Progamm-Ende "
  MENU TO wahl4
  IF wahl4 = 1                           && Programm-Ende gewählt
    CLEAR
    QUIT
  ELSE                                   && alle anderen Eingaben
    @ 2,20,4,36 BOX SPACE(9)             && bedeuten Wechsel im
    chk_arrow()                          && Hauptmenü
  ENDIF
RETURN
```

```
*
* Prozedur wird durch Rechtspfeil aufgerufen, wenn ein Unter-
* menü aktiv ist. Durch "künstliches" Esc wird das Untermenü
* verlassen und gleichzeitig der Merker für chk_arrow() ge-
* setzt.
*
PROCEDURE r_esc
PARAMETERS a, b, c
  flg = "R"
  KEYBOARD CHR(27)
RETURN

*
* Wie r_esc, jedoch für die Linkspfeil-Taste
*
PROCEDURE l_esc
PARAMETERS a, b, c
  flg = "L"
  KEYBOARD CHR(27)
RETURN

*
* Funktion prüft über den Kenner flg, ob ein Untermenü mit
* Rechts- oder Linkspfeil verlassen wurde. In diesem Fall
* erzeugt die Funktion eine Tastenfolge, die im Hauptmenü
* die Markierung nach rechts oder links um eine Stelle ver-
* schiebt und die gleichzeitig das entsprechende neue Unter-
* menü aktiviert. Außerdem wird .T. zurückgegeben, damit ge-
* prüft werden kann, ob die Untermenü-Anzeige gelöscht werden
* muß.
*
FUNCTION chk_arrow
  PRIVATE mov
  mov = .T.
  DO CASE
    CASE flg = "R"
      KEYBOARD CHR(4) + CHR(13)       && Rechtspfeil+Return zum
*                                        Aktivieren des näch-
*                                        sten Hauptmenüpunktes
*                                        rechts
```

```
    CASE flg = "L"
      KEYBOARD CHR(5) + CHR(13)        && Linkspfeil+Return zum
*                                         Aktivieren des näch-
*                                         sten Hauptmenüpunktes
*                                         links
    OTHERWISE
      KEYBOARD CHR(13)                 && Return zum Aktivieren
*                                      && desselben Untermenüs
      mov = .F.
  ENDCASE
  flg = " "                            && Merker zurücksetzen
RETURN(mov)
```

Dieses Programmbeispiel zeigt exemplarisch, wie man "Pull-Down" Menüs aufbauen kann. Natürlich müssen Sie anstelle der einfachen Ausgabe-Anweisungen in den Untermenüs Aufrufe "echter" Programm-Routinen einbauen. Interessant ist, daß durch die MENU TO-Anweisungen die Menü-Anzeige bei jedem neuen Aufruf automatisch neu geschrieben wird, so daß Sie den Bildschirm nicht zu sichern brauchen, wenn eine aufgerufene Prozedur den Bildschirm löscht oder ändert.

16.2 MENU TO "zweckentfremdet" zur Satzauswahl

Die folgende Prozedur verwendet PROMPT...MENU TO Anweisungen, um die Inhalte bestimmter Felder in Datensätzen anzuzeigen.

```
*
* Die Prozedur erwartet eine geöffnete Datenbank konten.dbf
* mit folgender Struktur:
*
* Feldname Feldtyp Zeichen Dez.
* ----------------------------
* KTONR       C        6
* BEZ         C       30
* usw.
*
PROCEDURE auswahl
PRIVATE spuffer, alt_sel, blank40, wahl, alt_satz, i, lky, ef

PRIVATE kto[20]                        && Speicher für Feld-
*                                         inhalte
```

```
SAVE SCREEN TO spuffer                  && Bildschirm sichern

alt_sel = select()                      && SELECT-Bereich sichern
SELECT konten                           && Kontendatei anwählen

blank40 = SPACE(40)                     && Leer-String

*
* Rahmen für Menü-Anzeige
*
@ 1,34,22,75  BOX CHR(201)+CHR(205)+CHR(187)+ ;
                  CHR(186)+"   "+CHR(186)+" "
@ 22,34,24,75 BOX CHR(204)+CHR(205)+CHR(185)+ ;
                  CHR(186)+CHR(188)+CHR(205)+ ;
                  CHR(200)+CHR(186)+" "
*
* Funktionstasten-Erläuterung anzeigen
*
@ 23,35 SAY "    Pfeiltasten PgUp PgDn Esc Return    "

GOTO TOP

wahl = 0

DO WHILE .T.
  alt_satz = RECNO()                    && aktuellen Satzzeiger
*                                          sichern
  FOR i = 1 TO 20                       && 20 Sätze einlesen,
    IF !EOF()                           && wenn nicht Datei-Ende
      kto[i] = ktonr + "     " + bez
      SKIP
    ELSE                                && sonst mit Leerstring
      kto[i] = blank40                  && füllen
    ENDIF
  NEXT
  SKIP                                  && prüfen, ob letzt. Satz
  ef = EOF()                            && erreicht wurde
  GOTO alt_satz                         && Satzzeiger zurück

  FOR i = 1 TO 20                       && 20 Menüpunkte anzeig.
    IF !empty(kto[i])
      @ 1+i,35 PROMPT kto[i]            && Konto-Nr. + Bezeichng.
```

```
    ELSE                                && oder
      @ 1+i,35 SAY blank40              && Leerfelder
    ENDIF
  NEXT

  MENU TO wahl                          && Kto-Nr. auswählen

  lky = LASTKEY()                       && Tastencode speichern,
*                                          mit dem MENU TO
*                                          verlassen wurde
*
* Tastendruck auswerten
*
  DO CASE
    CASE lky = 13                       && Return, dann Kto-Nr +
*                                          Return in den Tasta-
*                                          tur-Puffer "schieben"
      KEYBOARD ;
        SUBSTR(kto[wahl],1,6) + ;
        chr(13)
      EXIT                              && und Ende

    CASE lky = 18                       && PgUp = 20 Sätze zurück
      SKIP -20

    CASE lky = 3                        && PgDn = 20 Sätze vor,
      IF !ef                            && wenn nicht Dateiende
        SKIP 20
      ENDIF

    CASE lky = 27                       && Esc = Abbruch
      EXIT
  ENDCASE
ENDDO

RESTORE SCREEN FROM spuffer             && alten Bildschirm
*                                          zurückholen
SELECT(alt_sel)                         && alten Select-Bereich
*                                          anwählen
RETURN
```

Die Prozedur wurde in dieser Form in einem Buchhaltungsprogramm eingesetzt, um die in der Konten-Datei gespeicherten Konten anzuzeigen und eines davon auszuwählen.

Sie ist mit folgenden Anweisungen im Programm eingebunden:

```
SET KEY -9 TO kto_plan

*
* Anweisungen
*

kto1 = SPACE(6)
@ 12, 5 SAY "Kontonummer"
@ 12,18 GET kto1 PICTURE "999999"
READ

*
* Anweisungen
*

PROCEDURE kto_plan
PARAMETERS proz, zeile, var

IF proz = "KTO_PLAN"                    && rekursiven Aufruf
  RETURN                                && sperren
ENDIF

IF (SUBSTR(var,1,3) = "KTO") .AND. ;
   (proz = "BUCHEN")                    && Prüfen, ob Aufruf
  DO auswahl                            && im richtigen Programm-
ENDIF                                   && teil erfolgte

RETURN
```

Mit diesen Anweisungen wird erreicht, daß der Anwender immer dann, wenn das Programm die Eingabe einer Kontonummer erwartet, durch Drücken der Funktionastaste `F10` die Kontenauswahl anzeigen kann. Mit den Tasten `PgUp` und `PgDn` kann er in der Datei "blättern" und mit den Pfeiltasten eines der angezeigten Konten auswählen. Beim abschließenden Drücken von `Return` wird die Nummer des gewählten Kontos automatisch in das Eingabefeld für die Kontonummer übertragen und die Anzeige des Kontenplans gelöscht.

In ähnlicher Weise können Sie natürlich auch Prozeduren entwerfen, die andere Informationen lesen und anzeigen. Das Verfahren ist immer nützlich, wenn Sie dem Bediener einer Anwendung die Möglichkeit geben wollen, aus einem größeren Datenbestand Auswahlen zu treffen.

17 Die Clipper-Bibliothekdatei CLIPPER.LIB

Kernstück des Clipper Programmpaketes ist die Bibliothekdatei CLIPPER.LIB, in der alle Routinen enthalten sind, die ein compiliertes Clipper-Programm zum "Leben erwecken", denn aus dieser Datei werden durch den Linker alle Routinen in die ausführbare .EXE-Datei übertragen, die der Compiler angefordert hat.

Es ist einleuchtend, daß die Entwickler von Clipper den Inhalt von CLIPPER.LIB nicht veröffentlichen, da hier ein Großteil der Entwicklungsarbeit enthalten ist.

Allgemein ist es für den Entwickler von Clipper Programmen nicht notwendig, den Inhalt von CLIPPER.LIB zu kennen, da ja gerade die höhere Programmiersprache, in diesem Fall die erweiterte dBASE-Sprache, geschaffen wurde, damit der Programmierer nicht mühsam Programme in Assemblersprache schreiben muß.

In speziellen Fällen können jedoch einige Routinen in CLIPPER.LIB direkt aufgerufen werden, die nicht als Anweisungen oder Funktionen in der Clipper-Sprache angeboten werden.

Hier einige Beispiele:

17.1 Positionieren des Cursors _SETCURS

```
CLEAR
CALL __SETCURS WITH WORD(10), WORD(20)
?? "Test"
INKEY(0)
RETURN
```

Die interne Routine __SETCURS bringt den Cursor auf eine bestimme Bildschirm-Position. Die Routine erwartet zwei Parameter, die Zeile (0...24) und die Spalte (0...79) als Positionsangabe. Wie Sie dem Beispielprogramm entnehmen können, müssen diese beiden Parameter mit der WORD-Funktion übergeben werden. Das ist nötig, da die interne Routine die Parameter als 16-Bit Ganzzahlwerte erwartet, Clipper jedoch numerische Werte immer als Gleitpunktzahlen darstellt. Durch die WORD-Funktion erfolgt die Umwandlung.

Sie erkennen auch, daß interne Routinen mit der CALL-Anweisung aufgerufen werden, da hier die Parameter-Übergabe über den internen Stapelspeicher erfolgt und nicht wie bei UDFs über einen speziellen Benutzer-Stapelspeicher.

17.2 Cursor aus- und einschalten

Im Kapitel "Assembler-UDFs" finden Sie eine Routine, mit der verschiedene Cursor-Formen gesetzt werden können. Oft ist es jedoch nur nötig, den Cursor aus- bzw. einzuschalten, z.B. dann, wenn das Programm auf einen Tastendruck mit der INKEY()-Funktion wartet. Auch hierzu ein Beispiel:

```
CLEAR
? "Test"
CALL __SETCTYP WITH WORD(0)          && Cursor ausschalten
INKEY(0)
CALL __SETCTYP WITH WORD(1)          && Cursor einschalten
INKEY(0)
RETURN
```

Ab der Clipper-Version Sommer '87 stehen auch die Anweisungen SET CURSOR ON bzw. OFF zur Verfügung. Wie sich jedoch in der Praxis gezeigt hat, wird durch diese Anweisungen nicht in allen Fällen das interne Cursor-Statusflag richtig gesetzt. Dieses Statusflag wird von der ebenfalls in diesem Buch enthaltenen C-UDF is_curs() gelesen. Sie sollten also besser die oben gezeigten Aufrufe zum Ein- und Ausschalten des Cursors verwenden.

17.3 Rollen oder Löschen eines Bildschirm-Fensters

Mit der Bibliothek-Routine __SCROLL können Sie einen rechteckigen Bildschirm-Bereich auf- und abwärts rollen oder diesen Bereich löschen. Hier ein Beispiel:

```
CLEAR
n = 0
DO WHILE .T.
  FOR i=1 TO 9
    @ i,0 SAY REPLICATE(STR(i,1),80)
  NEXT
  @ 15,0 SAY "n: " GET n PICTURE "999"
  READ
  IF n = 999
    CLEAR
    QUIT
  ENDIF
  scroll(2,10,7,30,n)
  INKEY(0)
```

```
ENDDO

FUNCTION scroll
PARAMETERS zo,sl,zu,sr,n
  CALL __SCROLL WITH ;
                WORD(zo),WORD(sl),WORD(zu),WORD(sr), WORD(n)
RETURN(.T.)
```

In diesem Beispielprogramm ist zur Vereinfachung der Aufruf der Bibliothek-Routine __SCROLL in eine UDF (scroll()) "verpackt", damit man im Programm nicht ständig die Parameter mit der WORD()-Funktion angeben muß.

Die interne Routine __SCROLL erwartet fünf Parameter: Zeile (zo) und Spalte (sl) der oberen linken Ecke, Zeile (zu) und Spalte (sr) der rechten unteren Ecke eines Bildschirm-Ausschnittes sowie eine Zahl (n), die bestimmt, um wieviele Zeilen und in welcher Richtung der Ausschnitt zu rollen ist.

Bei positiven Werten von n wird der Bildschirm-Ausschnitt um die angegebene Zeilenzahl nach oben, bei negativen Werten um die entsprechende Zeilenzahl nach unten gerollt. Hat n den Wert 0, dann wird der Bildschirm-Bereich gelöscht.

17.4 Aktivieren eines Ausgabefensters

Die interne Routine __SETWIN möglicht einfache Fensterfunktionen, wie folgendes Beispiel zeigt:

```
CLEAR SCREEN
FOR i = 1 TO 2000
  ?? "*"
NEXT
INKEY(0)
CALL __SETWIN WITH WORD(5),WORD(10),WORD(20),WORD(30)
CLEAR SCREEN
FOR i = 1 to 100
  ? i
NEXT
@ 0,0 SAY "fertig..."
INKEY(0)
CALL __SETWIN WITH WORD(0),WORD(0),WORD(24),WORD(79)
```

```
FOR i = 1 to 100
  ? i
NEXT
INKEY(0)
```

Zunächst wird der gesamte Bildschirm mit Sternchen gefüllt. Danach legt das Programm durch Aufruf der Funktion __SETWIN ein neues Fenster fest, dessen Koordinaten von Zeile 5, Spalte 10 bis Zeile 20, Spalte 30 gehen. Das bedeutet, daß nachfolgende Ausgabe-Anweisungen nur noch in diesem Fenster erfolgen, wie durch die nachfolgenden Ausgabe-Anweisungen bestätigt wird.

Durch den erneuten Aufruf von __SETWIN mit den Werten 0, 0, 24, 79 wird wieder der gesamte Bildschirm als "Fenster" aktiviert.

17.5 Direkte Drucker-Ausgabe

Manchmal ist es umständlich, die Druckausgabe mit SET DEVICE TO PRINT oder mit SET PRINT ON zu aktivieren. Hier hilft die Bibliothek-Routine __POUT:

```
txt = "Das ist ein Test"
CALL __POUT WITH txt, WORD(LEN(txt))
```

Die Routine __POUT muß mit zwei Parametern aufgerufen werden, der auszugebenden Zeichenkette und der Länge der Zeichenkette.

__POUT ist die Druck-Ausgaberoutine der untersten Ebene, d.h. erfolgte Druck-Umleitungen mit SET PRINTER TO oder Anweisungen wie SET PRINTER OFF haben auf die Ausgabe keinen Einfluß. __POUT führt Druckausgaben direkt durch den Aufruf der entsprechenden DOS-Funktion aus.

17.6 Interne reservierte Wörter

Wie Ihnen sicher bereits aufgefallen ist, beginnen die Namen der internen Clipper-Routinen in der Regel mit einem doppelten Unterstreichungszeichen (_) oder mit dem Dollar-Zeichen ($). Wenn Sie in Ihrem Programm einen Variablenbezeichner, Prozedur- oder Funktionsnamen verwenden, der mit einem internen Namen identisch ist, dann wird es in fast allen Fällen zu einer Fehlfunktion des Programms kommen. Daher sollten Sie es sich zur Regel machen, keine von Ihnen definierten Namen mit Untersteichungs- oder Dollarzeichen zu beginnen.

Außerdem sollten Sie keine Namen benutzen, die mit denen von Clipper-Kommandos und Funktionen übereinstimmen. In der folgenden Liste finden Sie weitere, intern benutzte Namen, die Sie ebenfalls nicht verwenden dürfen (die Liste trifft auf die Clipper-Version Sommer ´87 zu) Sie enthält auch die in EXTEND.LIB definierten Namen.

$i4_8	$i4_ac_hi	$i4_ac_lo
$i4_add	$i4_cmp	$i4_div
$i4_divrev	$i4_errdiv0	$i4_errinf
$i4_errinvld	$i4_errminf	$i4_errpinf
$i4_fix	$i4_fixirnd	$i4_fixrnd
$i4_l	$i4_mul	$i4_norm
$i4_q	$i4_result	$i4_round
$i4_round_exp	$i4_round_flag	$i4_s
$i4_sub	$i4_subrev	$i4_to_l
$i4_to_lirnd	$i4_to_lrnd	$i4_to_q
$i4_to_qirnd	$i4_to_qrnd	$i4_to_s
$i4_to_sirnd	$i4_to_srnd	$i4_to_w
$i4_to_wirnd	$i4_to_wrnd	$i4_w
$i8_4	$i8_ac_hi	$i8_ac_lo
$i8_add	$i8_addf	$i8_addfsi
$i8_arg	$i8_clearac	$i8_cmp
$i8_c_pwr	$i8_div	$i8_divf
$i8_divfsi	$i8_divr	$i8_divrdi
$i8_divrev	$i8_errdiv0	$i8_errinf
$i8_errinvld	$i8_errminf	$i8_errpinf
$i8_errstack	$i8_even	$i8_exp
$i8_expadjust	$i8_exphrange	$i8_expm1
$i8_expm1reduced	$i8_exprange	$i8_fix
$i8_fixirnd	$i8_fixrnd	$i8_half
$i8_implicit_exp	$i8_input	$i8_input_ws
$i8_l	$i8_lgt	$i8_lnreduce
$i8_lnreduceac_hi	$i8_log	$i8_log_hi
$i8_log_lo	$i8_movac	$i8_movac_arg
$i8_movarg	$i8_movarg_ac	$i8_movtemp
$i8_mul	$i8_mulf	$i8_mulfsi
$i8_norm	$i8_one	$i8_output
$i8_overflowrange	$i8_pf_pwr	$i8_pf_pwr_l
$i8_poly	$i8_popac	$i8_poparg
$i8_popsi	$i8_pshac	$i8_psharg
$i8_pshsi	$i8_pwr	$i8_pwr_l
$i8_pzz	$i8_p_q	$i8_q

$i8_range	$i8_reduce	$i8_result
$i8_retone	$i8_retzero	$i8_round
$i8_round_exp	$i8_round_flag	$i8_s
$i8_sqr	$i8_sub	$i8_subf
$i8_subfsi	$i8_subr	$i8_subrdi
$i8_subrev	$i8_temp	$i8_tmul
$i8_to_l	$i8_to_lirnd	$i8_to_lrnd
$i8_to_q	$i8_to_qirnd	$i8_to_qrnd
$i8_to_s	$i8_to_sirnd	$i8_to_srnd
$i8_to_w	$i8_to_wirnd	$i8_to_wrnd
$i8_tpwr10	$i8_two	$i8_w
$i8_z	$i8_z2p_q	$i8_z3p_q
$i8_zz	ACHOICE	ACOPY
ADEL	ADIR	AFIELDS
AFILL	AINS	ALLTRIM
ALTD	AMPM	ASCAN
ASORT	BIN2I	BIN2L
BIN2W	BROWSE	CLEAR_GETS
CURDIR	DAYS	DBEDIT
DBF	DBFILTER	DBRELATION
DBRSELECT	DB_ERROR	DESCEND
DISKSPACE	DOSERROR	ELAPTIME
ERRORLEVEL	ERRORSYS	EXAMPLEP
EXPR_ERROR	FCLOSE	FCREATE
FERROR	FKLABEL	FKMAX
FOPEN	FREAD	FREADSTR
FSEEK	FWRITE	GETE
GET_PIC	HARDCR	HEADER
I2BIN	INDEXEXT	INDEXORD
ISALPHA	ISLOWER	ISPRINTER
ISUPPER	L2BIN	LEFT
LENNUM	LUPDATE	MEMOEDIT
MEMOLINE	MEMOREAD	MEMOTRAN
MEMOWRIT	MISC_ERROR	MLCOUNT
MLPOS	MOD	NETERR
NEXTKEY	OPEN_ERROR	OpTab
OS	PAD	PRINT_ERRO
PROCFILE	RAT	READEXIT
READINSERT	READKEY	RECSIZE
RESTSCREEN	RIGHT	SAVESCREEN
SCROLL	SECS	SETCANCEL
SETCOLOR	SETPRC	SOUNDEX

STATLINE	STRTRAN	STRZERO
STUFF	SUMMER87	TONE
TSTRING	UNDEF_ERRO	VERSION
XBROWSE	_brkctl	_BUFF_SIZE
_ceil	_dbf_delim	_dos_reserve
_edata	_end	_environ
_errno	_exit	_exp
_floor	_fmod	_free
_HUGE	_init_refs	_linelen
_log	_log10	_main
_malloc	_matherr	_name_2_code
_ntx_cold	_ntx_hold	_num_sout
_omit	_pow	_sqrt
_stccpy	_stpblk	_stpchr
_strcat	_strcpy	_strlen
_strncpy	_translate	_VALL
_VREPORT	_VSORT	_VSTD
__abrkp	__abrktb	__abrktbe
__ABS	__ACCEPT	__ACCUM
__acfinfo	__acrtmsg	__acrtused
__adbgmsg	__ADD_FIELD	__ADJ2000
__aexit_rtn	__aFahdiff	__aFFaldiv
__aFFalmul	__aFFaulmul	__aFFbldiv
__aFFblmul	__aFFbulmul	__aFldiv
__aFlmul	__aFlrem	__aFlshl
__aFlshr	__aFNaldiv	__aFNalmul
__aFNaulmul	__aFNbldiv	__aFNblmul
__aFNbulmul	__aFuldiv	__aFulmul
__aFulrem	__ahdiff	__AHINCR
__AHSHIFT	__aintdiv	__alloc_reef
__altc_error	__altc_on	__alternate_on
__alt_handle	__alt_open	__amalloc
__amallocbrk	__amblksiz	__amexpand
__amlink	__amsg_exit	__AND
__APPEND	__ARRAY	__ARROW
__ASC	__asegl	__asegds
__asegh	__asegn	__asegr
__asizds	__asmcall	__astart
__AT	__atod	__aton
__atopsp	__attach_reef	__BACK
__bcmp	__bcopy	__bcopyf
__BEGIN_SEQ	__bell_on	__BFIL2

__BFILD	__BFIS2	__BFIST
__BFUST	__bgetint	__bilge
__blk_addr	__blk_alloc	__blk_end
__blk_grow	__blk_init	__blk_max
__blk_release	__blk_shrink	__bmove
__BOF	__BORDER	__BOX
__bputint	__BREAK	__break_cont
__break_cycle	__BRKSTAT	__bscan
__bset	__buf_alloc	__buf_release
__CALL	__carry_on	__CCLR
__CDOW	__century_on	__CFLTCVT
__cfltcvt_tab	__CFTOE	__CFTOF
__CFTOG	__child	__CHK_PC
__chk_sym	__CHR	__chunk
__ch_type	__CIN	__cinit
__cintDIV	__close_dbf	__CLRALL
__CLREOL	__CLREOS	__CLR_RELATE
__CMONTH	__COL	__COLOFF
__COLORD	__confirm_on	__console_on
__COUT	__CROPZEROS	__CSTAT
__ctermsub	__CTOD	__ctrandisp1
__ctrandisp2	__curs_on	__C_ALL
__C_DBF	__C_EOL	__C_EOS
__C_GETS	__C_SCREEN	__DATE
__date_mode	__DAY	__day_name
__dbf	__dbf1_sel	__dbf2_sel
__dbf3_sel	__dbfhead	__dbfopen
__DBF_CLOSE	__DBF_COMMIT	__DBF_COPY
__DBF_CREATE	__dbf_dir	__dbf_mode
__DBF_OPEN	__DBF_SELECT	__DBF_SET
__DBF_STRUX	__dbgo_filter	__dbgo_relate
__DBXBAG	__db_break	__db_goto
__db_init	__db_quit	__db_reef
__dcvt	__dcvtdisi	__dcvtst0
__dcvtst0a	__debug	__debug_active
__debug_force	__decimals	__DECLARE
__default_drive	__DELETE	__deleted_on
__delimiters_on	__DEL_FLAG	__DEL_MARK
__derr	__detach_reef	__device_p
__dinfo	__DIVIDE	__dmovtmpesbx
__dmovtmpessi	__dmytod	__DO
__dosall	__doserrno	__dosfree

__dosfunc	__dosvermajor	__dosverminor
__DOS_CALL	__dos_hold	__DOW
__dspace	__dtoa	__DTOC
__DTOS	__dtox	__dvabs
__dvadd	__dvdiv	__dveq
__dveqz	__dvexp	__dvfma
__dvfmi	__dvfml	__dvfmq
__dvge	__dvgt	__dvinit
__dvint	__dvle	__dvlog
__dvlt	__dvltz	__dvmod
__dvmul	__dvne	__dvneg
__dvpow	__dvrnd	__dvsqr
__dvsub	__dvtoi	__dvtol
__dvtoq	__DV_MIN_LONG	__DV_NEG_MIN_LONG
__eaddd	__eadds	__EBACK
__edivd	__edivdr	__edivs
__edivsr	__EEQ	__EFORE
__EJECT	__eldd	__eldl
__elds	__eldw	__emm_handle
__emm_stat	__EMPTY	__emuld
__emuls	__ENDPROC	__END_SEQ
__EOF	__EQ	__escape_on
__estd	__estdp	__ests
__estsp	__esubd	__esubdr
__esubs	__esubsr	__eval
__EVENTS	__exact_on	__exclu_on
__exit	__exmback	__exmgrab
__EXPON	__e_avail	__fabs
__fac	__fadd	__faddd
__fadds	__FALIAS0	__FALIAS1
__FALSE	__FASSIGN	__fault_alloc
__fchs	__fcmp	__fcompp
__FCOUNT	__fcsp	__fctmp
__fctopst	__fdiv	__fdivd
__fdivdr	__fdivr	__fdivs
__fdivsr	__fdup	__ffree
__FF_MSGBANNER	__FIELDNAME	__FILE
__find_field	__fixed_on	__fixlen
__FKEYCNT	__fkeys	__fldd
__fldl	__flds	__fldt
__fldw	__fldz	__FLOCK
__fltin	__fltinf	__fltout

__fltused	__fmalloc	__fmt_sym
__fmul	__fmuld	__fmuls
__FORCDECPT	__form_reef	__FOUND
__fpinit	__fpmath	__fpsigadr
__fpsignal	__fptrap	__FSELECT0
__FSELECT1	__fstd	__fstdp
__fsts	__fstsp	__fstt
__fsttp	__fsub	__fsubd
__fsubdr	__fsubr	__fsubs
__fsubsr	__ftol	__ftst
__FUNC	__fv_alloc	__fv_release
__fxch	__fxchq	__f_first
__f_next	__GE	__GET
__GETCTYP	__GETCURS	__getenv
__getpfba	__get_name	__GET_RANGE
__get_reef	__GET_SP	__get_sym
__GET_VALID	__gl	__GOTO
__GOTO_BOTT	__goto_db	__GOTO_TOP
__gout	__goutff	__goutnl
__go_self	__GT	__heap_init
__heap_stat	__hold_stream	__horror
__icreate	__ic_out	__ierror
__IKEY	__IMODE	__indext
__INKEY0	__INKEY1	__INPUT
__insert_mode	__INSTR	__INT
__intensity_on	__intno	__ISCOLOR
__JEOF	__KEND	__KEYBOARD
__keymgr	__keyset	__keysub
__key_save	__key_update	__KHANDLE
__kill_ntx	__kill_reef	__KINIT
__LASTKEY	__LASTREC	__last_error
__last_file	__LBL	__LBL_BEGIN
__LBL_END	__ldelim	__LE
__LEN	__LINE	__line_num
__LKEY	__lntoa	__locase
__LOCATEP	__locate_key	__LOCK
__LOGQ	__LOWER	__LT
__ltable	__LTRIM	__mac_execute
__mac_exec_aug	__mac_expand	__mac_init
__mac_release	__mac_valid	__mappage
__margin	__matherr_flag	__MAX
__MAXCOL	__MAXROW	__mbp

__mcount	__MEMORY	__mem_alloc
__mem_release	__mem_snap	__MENU
__merge	__message	__MIN
__MINUS	__MODULUS	__MONTH
__month_name	__mout	__moutn
__MPLANKTON	__mstack	__mtop
__MULT	__mv_alloc	__mv_kill
__myalloc	__m_avail	__M_RESTORE
__M_SAVE	__nbuff	__NE
__NEGATE	__netalck	__netaunl
__neterr	__netflck	__netfunl
__NETNAME	__netrlck	__netrunl
__netwho	__new_str0	__new_str_stack
__NEXT	__next_key	__next_sym
__nfile	__nfree	__nmalloc
__NMSG_TEXT	__NMSG_WRITE	__node_search
__NOT	__NO_SNOW	__ntoa
__ntx_blks	__NTX_CLOSE	__NTX_COMMIT
__NTX_CREATE	__ntx_end	__ntx_grow
__ntx_init	__NTX_KEY	__NTX_OPEN
__NTX_ORDER	__ntx_put_header	__ntx_shrink
__NULL	__nullcheck	__num_blks
__ONE1	__openpid	__open_file
__OR	__oserr	__osfile
__osmajor	__osminor	__osmode
__osversion	__os_copy	__os_dir
__os_type	__ovlflag	__ovlvec
__PACK	__page_append	__page_error
__page_flush	__page_read	__page_update
__page_write	__PARAM	__param_count
__parc	__parclen	__parcsiz
__pards	__parinfa	__parinfo
__parl	__parnd	__parni
__parnl	__path	__pcmp
__PCOL	__PCOUNT	__pcycle
__PEND	__PERROR	__pfilled
__pgmptr	__PHANDLE	__pid_cls
__PINIT	__PLANKTON	__PLUS
__POP	__POPF	__POPM
__POPM2	__POSITIVE	__POS_XY
__POUT	__PRINT	__PRINTNL
__PRINTS	__print_on	__PRIVATE

__PROCLINE	__PROCNAME	__PROMPT
__PROW	__psp	__pspadr
__ptop	__ptrnorm	__ptr_add
__PUBLIC	__pull1	__pull2
__purge_reef	__PUSHA	__PUSHM
__PUSHP	__PUSHS	__PUSHV
__push_unit	__putcs	__putcsl
__putd	__putdate	__putl
__putln	__PUTM0	__putn
__putq	__putsym	__putw
__pv_alloc	__p_handle	__QChdata
__QENVIRON	__QEXP	__QSQRT
__quick_sort	__QUIT	__quitting
__rdelim	__READ	__READVAR
__read_delim	__read_memo	__read_reef
__read_sdf	__read_sdreef	__read_var
__RECALL	__RECNO	__REEF
__reef_break	__reef_init	__reef_range
__reef_scope	__reef_size	__REINDEX
__RELEASE	__REP	__REPLICATE
__REP_BEGIN	__REP_END	__ret
__retc	__retclen	__retds
__retl	__retnd	__retni
__retnl	__REWIND	__rl_break
__ROUND	__ROW	__ROWOFF
__rterror	__saddd	__sadds
__salt	__salterr	__saltmem
__saltprn	__salt_alias	__salt_aug
__salt_info	__salt_memlen	__salt_name
__salt_sym	__SAVE_RET	__SAY
__SBACK	__SBOX	__scann
__scanw	__scan_filename	__SCOPE_DEC
__SCOPE_SET	__score_on	__SCROLL
__SCRREST	__SCRSAVE	__SCRSIZE
__SCR_RESTORE	__SCR_SAVE	__sdivd
__sdivdr	__sdivs	__sdivsr
__SECONDS	__SEEK	__seek_db
__SELECT	__selection	__SEND
__setargv	__SETATTR	__SETCTYP
__SETCURS	__SETDPTR	__SETENH
__setenvp	__SETFKEY	__SETSTD
__setting	__SETUNS	__SETWIN

__SET_ALTER	__SET_COLOR	__SET_DATE
__SET_DECIMALS	__SET_DEFAULT	__SET_DELIM
__SET_DEVICE	__SET_FILTER	__SET_FMT
__SET_FUNCTION	__SET_KEY	__SET_LOCATE
__SET_MARGIN	__SET_OFF	__SET_ON
__SET_PATH	__SET_PC	__SET_PRINTER
__SET_RELATE	__SET_TA	__sfcount
__sfields	__SFORE	__SHANDLE
__shark_bait	__SINIT	__sizeof_memo
__SKIP	__slbase	__sldd
__sldl	__slds	__sldt
__sldw	__smovtmpesbx	__smovtmpessi
__smuld	__smuls	__snap
__snap_reserve	__snap_total	__softseek
__SORT	__SORT_BEGIN	__sort_cleanup
__SORT_END	__sort_out	__SOUT
__SPACE	__sptoq	__SPUSHM
__srbase	__srsize	__sstd
__sstdp	__ssts	__sstsp
__sstt	__ssttp	__ssubd
__ssubdr	__ssubs	__ssubsr
__stack_init	__stk_get	__STR1
__STR2	__STR3	__str_alloc
__str_release	__SUB2	__SUB3
__SYMBOL	__sym_count	__sym_init
__sym_search	__syscall	__system
__sys_date	__s_order	__tbuff
__tclose	__tcmp	__tcommit
__tcreat	__tctemp	__tend
__term_end	__term_init	__terror
__thcount	__TIME	__tinit
__tlock	__tlseek	__tmname
__topen	__tos	__TO_DEST
__to_file	__to_handle	__to_print
__TPLANK	__TRANS	__trans_reef
__trap_break	__trap_shark	__trash
__tread	__trename	__TRIM
__TRUE	__try_allocn	__tunlink
__tversion	__twrite	__TYPE
__umaskval	__unique_on	__UNLOCK
__upcase	__UPDATED	__UPPER
__USED	__VAL	__WAIT

__WAVE	__WAVEA	__WAVEL
__WAVEP	__weedbed	__weedend
__WINREST	__WINSAVE	__WORD
__wrap_on	__write_delim	__write_memo
__write_reef	__write_sdf	__write_sdreef
__wrt2err	__WSET	__WSYMBOL
__xaccum	__XBACK	__XFORE
__xframe0	__xframe1	__xframe2
__xframe3	__xnxt	__xpcol
__xpopf	__xpopm	__xprow
__xpushf	__xpushm	__XRELEASE
__xrew	__xsetcolor	__xtrans
__xtype	__YEAR	__ynstr
__ZAP	__ZER0	__zindex
___argc	___argv	

18 Verwalten der selbst definierten Routinen

Wenn Sie eine Weile Programme mit Clipper geschrieben haben, werden Sie zwangsläufig eine Vielzahl von Prozeduren und Funktionen erstellt haben, die Sie auch in anderen, neu zu entwickelnden Anwendungen gut verwenden können, besonders dann, wenn diese Prozeduren und Funktionen so entworfen wurden, daß sie möglichst universell eingesetzt werden können.

Es gibt nun mehrere Möglichkeiten, diese Routinen zu verwalten.

1. Als Quellcode

Es ist klar, daß man den Quellcode der Routinen aufbewahren sollte. Sie können nun in einem neuen Programm die Routinen verwenden, indem Sie mit dem Text-Editor die Quellcodes der benötigten Funktionen in den Quelltext der neu zu erstellenden Applikation hineinkopieren. Dieses Verfahren ist umständlich und zeitaufwendig, zumal die ja bereits getesteten Routinen jedesmal durch den Compiler neu übersetzt werden müssen. Der einzige Vorteil ist vielleicht, daß in einem Ausdruck des Quellcodes auch automatisch diese Routinen enthalten sind.

2. Als einzelne Objekt-Module

Hierzu muß jede einzelne Routine in einer getrennten Quelldatei gespeichert sein, die Sie einmal mit Clipper übersetzen. So erhalten Sie für jede Routine eine getrennte OBJ-Datei. Beim Linken des Anwendungsprogramms müssen Sie zusätzlich die Namen der benötigten OBJ-Dateien angeben. Dieses Verfahren birgt die Gefahr, daß man vergißt, OBJ-Dateien anzugeben und daß dadurch der Link-Lauf wiederholt werden muß. (Sie erhalten entsprechende Fehlermeldungen vom Linker.)

3. Als Gesamt-Objekt-Modul

Hierzu müssen Sie die Quellcodes aller Routinen in einer PRG-Datei zusammenfassen und diese einmal compilieren. Danach steht Ihnen eine OBJ-Datei zur Verfügung, die Sie beim Linken der Anwendung mit angeben müssen. Damit ist die Sicherheit gegeben, daß keine Routine beim Linken vergessen wird. Aber auch hier gibt es einen Nachteil: der Linker bindet den gesamten, in einer OBJ-Datei enthaltenen Code zum Anwendungsprogramm, selbst dann, wenn in der OBJ-Datei Routinen vorhanden sind, die in dieser konkreten Anwendung nicht benötigt werden. Das vergrößert jedoch die erzeugte EXE-Datei. Somit ist dieses Verfahren nur praktikabel, wenn die Routinen in der OBJ-Datei (fast) alle in der Anwendung benötigt werden.

4. Als Bibliothek-Datei (Library)

In einer Library ist eine beliebige Anzahl von Routinen zusammengefaßt. Der Linker sucht beim Binden eines Programms *nachdem* er alle OBJ-Dateien gelesen hat, die Libraries ab und lädt aus ihnen alle noch fehlenden Routinen. Damit ist sichergestellt, daß zur EXE-Datei nur die Routinen gebunden werden, die auch tatsächlich erforderlich sind.

Wichtig ist, daß die Libraries erst durchsucht werden, *nachdem* die OBJ-Dateien gelesen wurden. Somit ist es möglich, eine Library-Routine zu überlagern, wenn in einer OBJ-Datei eine gleichnamige Routine enthalten ist; diese erhält den Vorzug. Sie wenden dieses Verfahren immer dann an, wenn Sie zum Test das Modul DEBUG.OBJ mit einbinden. In diesem Fall enthält DEBUG.OBJ den Code des Debuggers, während beim Binden ohne DEBUG.OBJ aus CLIPPER.LIB eine "Blindroutine" DEBUG gelesen wird.

Eigene Routinen in einer oder mehreren Libraries zusammenzufassen, hat den Vorteil, daß zur Anwendung nur die benötigten Routinen gebunden werden und daß Sie beim Aufruf des Linkers lediglich den oder die Library-Namen anzugeben brauchen.

18.1 Das Anlegen und Verwalten eigener Bibliothek-Dateien (Libraries)

Wenn Sie eigene Libraries anlegen und verwalten möchten, benötigen Sie ein Hifsprogramm, den sogenannten "Library-Manager", wie z.B. das Programm LIB.EXE von MICROSOFT, das zusammen mit fast allen MICROSOFT-Produkten ausgeliefert wird.

Bevor Sie eine neue Library aufbauen, müssen Sie die gewünschten Routinen compilieren. Es ist sinnvoll, dazu für jede Routine eine PRG-Datei zu bilden, die dann einzeln compiliert wird. Danach haben Sie also für jede Routine eine einzelne OBJ-Datei. Es muß sich dabei nicht nur um Dateien handeln, die mit Clipper erzeugt wurden. Sie können auch OBJ-Dateien verwenden, die mit dem C-Compiler oder mit einem Assembler erstellt wurden. Diese OBJ-Module müssen natürlich so gebildet sein, daß sie zu Clipper-Modulen "passen". (Siehe die Kapitel über UDFs in anderen Sprachen in diesem Buch.)

Das *Bilden einer neuen Library* erfolgt mit dem Kommando

```
LIB <LIB-Name> +<Modul1> +<Modul2> ... +<Moduln>;
```

LIB-Name bezeichnet den Namen der neu zu bildenen Library (ohne den Zusatz .LIB) und $Modul_1$, $Modul_2$ usw. geben die Namen der betreffenden OBJ-Module an (ohne den Zusatz .OBJ).

Beispiel:

```
LIB CLGRAF +G_BOX +G_CLS +G_DOT +G_HGR +G_LINE +G_MODE +G_PAL;
```

Dieses Kommando bildet eine Library CLGRAF.LIB, in der die Routinen G_BOX, G_CLS usw. enthalten sind. (Es handelt sich hierbei um die Objekt-Module der in diesem Buch beschriebenen Grafik-Routinen für Clipper.)

An eine bestehende Library können Sie auch *neue Module anfügen.* Dazu dient folgendes Kommando:

```
LIB <LIB-Name> +<Modul1> +<Modul2> ... +<Moduln>;
```

Das Kommando ist identisch mit dem zum Bilden einer neuen Library. Hier wird als LIB-Name lediglich der Name einer bereits existierenden Library angegeben. Der Lib-Manager fügt die neuen Module ein und legt automatisch eine Kopie der "alten" Library mit dem Zusatz .BAK an.

Wenn Sie *Module* aus einer Library *entfernen* möchten, geben Sie das Kommando

```
LIB <LIB-Name> -<Modul1> -<Modul2> ... -<Moduln>;
```

Das Minuszeichen vor den Namen der OBJ-Module kennzeichnet, daß diese entfernt werden sollen.

Zum *Austauschen von Modulen* in einer Library gilt folgendes Kommando:

```
LIB <LIB-Name> -+<Modul1> -+<Modul2> ... -+<Moduln>;
```

Durch dieses Kommando werden die angegebenen OBJ-Module aus der Library entfernt und durch gleichnamige, auf der Diskette oder Platte gespeicherte Module ersetzt.

Der *Inhalt* einer Library kann in eine Textdatei *ausgegeben* werden. Dafür geben Sie das Kommando

```
LIB <LIB-Name>,<List-Datei>
```

Hierdurch wird eine Text-Datei angelegt, in der eine Referenzliste enthalten ist, die die in der Library enthaltenen Module beschreibt.

Mit diesen Grund-Anweisungen können Sie Ihre eigenen Bibliothek-Dateien erzeugen und verwalten. Der Lib-Manager verfügt noch über eine Reihe weiterer Kommandos, die Sie bitte der Hersteller-Dokumentation entnehmen.

Tip: Zusammen mit Clipper werden die Bibliothek-Dateien CLIPPER.LIB und EXTEND.LIB ausgeliefert. Die Aufteilung in zwei Dateien ist erforderlich, da eine einzelne Datei größer als 360 KByte wäre und somit nicht mehr auf einer "normalen" IBM-Diskette Platz hätte. Wenn Sie jedoch mit einer Festplatte arbeiten, ist es nützlich, EXTEND.LIB mit CLIPPER.LIB zu kombinieren. Sie brauchen dann beim Link-Aufruf nicht zwei .LIB-Dateien anzugeben. Auch dazu kann der Library-Manager verwendet werden:

```
LIB CLIPPER +EXTEND.LIB;
```

Beachten Sie, daß der Zusatz .LIB hinter EXTEND unbedingt angegeben werden muß, da LIB andernfalls den Zusatz .OBJ annimmt.

19 Was sonst nirgendwo hineinpasst...

In diesem Kapitel sind Tips und Tricks zusammengefaßt, die sich nicht so recht unter einem anderen Oberbegriff einordnen lassen. Die hier angesprochenen Punkte sollen auch dazu dienen, Ihnen Denkanstöße für eigene Ideen zu geben.

19.1 "Mehrdimensionale" Arrays

Clipper bietet neben einfachen Speichervariablen auch eindimensionale Arrays. Im Gegensatz zu anderen Programmiersprachen sind die Clipper-Arrays recht flexibel, denn jedes Element eines Arrays kann hier unterschiedliche Datentypen speichern. Andere Sprachen bieten hingegen nicht nur ein-, sondern mehrdimensionale Arrays. Wem das Denken in mehreren Dimensionen leichter fällt, kann auch in Clipper geholfen werden: schreiben Sie einfach eine Funktion, die mehrere Dimensionen in eine Dimension projiziert.

Was sich kompliziert liest, sieht in einem Beispiel einfach aus:

```
*
* Demoprogramm für ein "zweidimensionales" Array, 2 * 5
*

CLEAR
DECLARE a[10]

a[1]  = "a11"
a[2]  = "a12"
a[3]  = "a13"
a[4]  = "a14"
a[5]  = "a15"
a[6]  = "a21"
a[7]  = "a22"
a[8]  = "a23"
a[9]  = "a24"
a[10] = "a25"

FOR x = 1 TO 2
  FOR y = 1 TO 5
    ? array2(x,y)
  NEXT
NEXT
```

```
FUNCTION array2
PARAMETERS x, y
RETURN a[(x-1)*5 + y]
```

Dieses Beispielprogramm demonstriert, wie ein eindimensionales Array mit 10 Elementen mit einer Anwender-definierten Funktion (`array`) in ein zweidimensionales Array mit 2 mal 5 Elementen transformiert wird.

Wenn Sie wollen, können Sie so Arrays mit nahezu beliebig vielen Dimensionen simulieren, allerdings ist es recht schwierig, sich ein Array mit sieben Dimensionen bildlich vorzustellen.

19.2 Papier-Sparroutine

Besonders in der Testphase eines Programms, das viele Ausdrucke produziert, wird viel Zeit darauf verwendet, Probe-Ausdrucke anzufertigen, die dann doch noch Fehler aufweisen. Nach erfolgter Korrektur muß ein erneuter Testlauf mit weiteren Ausdrucken erfolgen usw.

Neben dem unnötigen Papierverbrauch schlägt dabei die wertvolle Programmiererzeit negativ zu Buche.

Die folgenden Anweisungen helfen Ihnen sparen:

```
*
* Anfang des Hauptprogramms
*
PUBLIC dr_flag
dr_flag = .F.
SET KEY -4 TO druck_ea

*
* Programmtext
*
```

```
*
* Beginn eines Ausdrucks
*
IF ISPRINTER() .OR. dr_flag
  *
  * Ausdrucken
  *
ELSE
  *
  * Drucker-nicht-bereit-Meldung und Fehlerbehandlung
  *
ENDIF

PROCEDURE druck_ea
PARAMETERS proz, zeile, var
IF jn_meldung(20,"Druckausgabe in Datei")
  SET PRINTER TO temp.prn
  dr_flag = .T.
ELSE
  SET PRINTER TO
  dr_flag = .F.
ENDIF
RETURN
```

In diesem Programmbeispiel wird die Funktionstaste F5 dazu definiert, die Prozedur `druck_ea` aufzurufen. In dieser Prozedur wird die Druckausgabe in eine Datei (`temp.prn`) "umgeleitet", wenn der Anwender die Abfrage mit "J" beantwortet. Bei Eingabe von "N" wird die "normale" Druckausgabe wieder aktiviert.

Die Prozedur steuert gleichzeitig eine logische Variable `dr_flag`, die dazu verwendet werden kann, die Drucker-Bereitprüfung mit `ISPRINTER()` zu umgehen, wie Sie auch im obigen Beispiel sehen.

Sie können natürlich die Prozedur `druck_ea` so ergänzen, daß der Anwender den Dateinamen der Druckdatei angeben kann. Beachten Sie, daß dabei keine GET...READ-Anweisung verwendet werden sollte, da sonst möglicherweise READ-Anweisungen im aufrufenden Programm gestört werden.

Die Anweisung `SET PRINTER TO` ist primär zum Einsatz unter einer Netzwerk-Umgebung gedacht, aber wie Sie aus diesem Beispiel ersehen, kann sie auch in einer Einplatz-Anwendung sinnvoll verwendet werden.

Da nach `SET PRINTER TO <datei>` alle Druckausgaben (`?`, `??`, `@...SAY`) in die Datei erfolgen und auch alle Drucker-Steuerzeichen übertragen werden, kann es auch in einer "normalen" Anwendung sinnvoll sein, die Druck-Ausgabe zuerst in eine Datei zu leiten und diese dann später auszudrucken. Dazu ist das DOS-Kommando `PRINT` oder ein anderes Hintergrund-Druckprogramm gut geeignet, da dann in der Zwischenzeit weitergearbeitet werden kann.

19.3 Programm-Überwachung und Sicherheit

Wenn Sie Anwendungen entwickeln, die von dritten Personen bedient werden sollen, so ist es fast selbstverständlich, daß diese Programme möglichst sicher gegen Fehlbedienungen sein müssen. Das gilt ganz speziell, wenn Sie bei den Bedienern keine EDV-Fachkenntnisse voraussetzen können.

Gegen eine Fehlbedienung können Sie allerdings keinen Schutz einbauen: das "unsaubere" Verlassen des Programms. Das kann dadurch erreicht werden, daß man den Computer ausschaltet oder neu startet, obwohl das Anwendungsprogramm *nicht* über das normal vorgesehene Programm-Ende verlassen wurde.

Besonders bei Anwendungen, die Massendaten bearbeiten, kann dieser "Brutal-Ausstieg" schlimme Folgen haben, wenn Dateien nicht richtig geschlossen werden. Die Folge sind meistens fehlerhafte oder fehlende Daten, korrupte Index-Dateien oder sogar in ihrer Struktur beschädigte Datenbanken, die keinen ordnungsgemäßen Zugriff mehr gestatten.

Bedingt durch die Arbeitsweise des Betriebssystems, werden Informationen, die in den Satz einer Datei geschrieben werden, zunächst in einen Puffer übertragen, jedoch nicht unbedingt sofort physisch auf die Platte kopiert.

Der Puffer wird erst dann wirklich geschrieben, wenn er voll ist, bzw. wenn die Datei an eine andere, genügend weit entfernte, Stelle positioniert wird.

Die beste Garantie, daß alle Daten physisch geschrieben werden, ist die, die Datei zu schließen.

Bei Clipper (und auch bei dBASE) kommt hinzu, daß erst beim Schließen der Kopfsatz (Header) einer Datenbank aktualisiert wird, so daß erst dann gewährleistet ist, daß alle Informationen geschrieben und wieder auffindbar sind.

Aus dieser Tatsache kann man entnehmen, daß es für die Sicherheit der Datenbestände wichtig ist, eine Datei nur solange geöffnet zu halten, wie es für die Anwendung erforderlich ist. Das bedeutet, daß man *nicht* zu Beginn des Programms alle im gesamten Programm benötigten Dateien schon einmal "auf Vorrat" öffnet. Stattdessen sollte man zu Anfang jeder Teilfunktion die Dateien öffnen, die konkret benötigt werden, und diese beim Verlassen der Funktion wieder schließen. Die dadurch bedingte, kleine Verzögerung kann im Sinne einer größeren Sicherheit immer akzeptiert werden.

Ab der Clipper-Version Sommer '87 ist die Anweisung COMMIT verfügbar, mit der ein physisches Schreiben aller Dateien auf die Platte ausgelöst werden kann. Allerdings wird die Anweisung erst ab DOS-Version 3.3 unterstützt, so daß die obigen Angaben für viele Systeme nach wie vor gültig sind.

Sehr häufig kommt es vor, daß eine Anwendung beim Kunden lange fehlerfrei arbeitet und eines Tages kommt der Hilferuf "Nichts geht mehr! Alle Daten sind futsch. Ihr Programm hat einen Fehler!"

Wenn der Kunde das getan hat, was Sie ihm (hoffentlich) geraten haben, nimmt er seine Datensicherung vom Vortag, spielt diese ein - und alles geht wieder. Somit sind wenigstens nicht alle Daten verlorengegangen.

Die Frage ist natürlich, wie es zu den fehlerhaften Dateien kommen konnte. Wenn man einmal einen Programmfehler ausschließt, der nur bei bestimmten Kombinationen von Daten oder Bedienungsabläufen auftritt, so ist in fast allen Fällen ein durch den Kunden verursachter "unsauberer" Programmausstieg schuld - nur das wird er nicht zugeben.

Es liegt an Ihnen, den Beweis zu führen...

Die folgenden Routinen helfen Ihnen dabei:

```
* Eintrag in eine Log-Datei vornehmen (Start)
* und diese ggf. anlegen
*
PROCEDURE logein
PRIVATE i, rn, maxrec

IF !FILE("log.dbf")
  maxrec = 10        && hier Anzahl der gewünschten Sätze in
*                       der Log-Datei einsetzen
```

```
  CREATE stru
  APPEND BLANK
  REPLACE field_name WITH "SDATUM"
  REPLACE field_type WITH "D"
  REPLACE field_len WITH 8
  REPLACE field_dec WITH 0
  APPEND BLANK
  REPLACE field_name WITH "STZEIT"
  REPLACE field_type WITH "C"
  REPLACE field_len WITH 5
  REPLACE field_dec WITH 0
  APPEND BLANK
  REPLACE field_name WITH "EDATUM"
  REPLACE field_type WITH "D"
  REPLACE field_len WITH 8
  REPLACE field_dec WITH 0
  APPEND BLANK
  REPLACE field_name WITH "EZEIT"
  REPLACE field_type WITH "C"
  REPLACE field_len WITH 5
  REPLACE field_dec WITH 0
  CREATE log FROM stru
  ERASE stru.dbf
  USE log
  APPEND BLANK
  REPLACE stzeit WITH "    2"          && erste Satz-Nr. für
*                                        Log-Eintrag
  REPLACE ezeit WITH STR(maxrec,5)     && max. Anzahl Einträge
  FOR i = 1 TO maxrec                  && entsprechend viele
    APPEND BLANK                       && leere Sätze anfügen
  NEXT
  CLOSE DATABASES
ENDIF

*
* Log-Eintrag (Programm-Start)
*
USE log
GO TOP
rn = VAL(stzeit)                       && Nr. des Satzes für den
*                                        nächsten Eintrag
```

```
maxrec = VAL(ezeit)                    && max. Anzahl Einträge
GO rn                                  && Auf den Satz für den
*                                         Eintrag positionieren
REPLACE sdatum with DATE()             && Start-Datum und
REPLACE stzeit WITH ;
             SUBSTR(TIME(),1,5)        && Zeit
REPLACE edatum WITH CTOD("  .  .  ")   && End-Datum und -Zeit
REPLACE ezeit WITH SPACE(5)            && löschen

GO TOP                                 && Satz-Nr. für nächsten
rn = rn + 1                            && Eintrag festlegen
IF rn > maxrec                         && Maximalzahl erreicht ?
  rn = 2                               && wieder zum Anfang
ENDIF
REPLACE stzeit WITH STR(rn,5)          && speichern
CLOSE DATA
RETURN
*
* Eintrag in eine Log-Datei vornehmen (Ende)
*
PROCEDURE logaus
PRIVATE rn, maxrec
USE log
GO TOP
maxrec = VAL(ezeit)
*                                         Nummer des aktuellen
*                                         Satzes lesen
rn = IF((VAL(stzeit)-1)=1, maxrec-1, VAL(stzeit)-1)
GO rn
REPLACE edatum WITH DATE()             && End-Datum und
REPLACE ezeit WITH ;
                SUBSTR(TIME(),1,5)     && Zeit eintragen
CLOSE DATA
RETURN
```

Diese beiden Prozeduren verwalten eine Log-Datei, in die der Programmstart und das Programmende jeweils mit Datum und Zeit eingetragen werden. Damit diese Log-Datei nicht unbegrenzt anwächst, kann durch die Vorbelegung der Variablen `maxrec` bestimmt werden, wieviele Sätze für Einträge bereitgestellt werden sollen. Wenn entsprechend viele Eintragungen vorgenommen wurden, beginnt der Eintrag wieder ab Satz 2 der Datei.

Der erste Satz wird nicht für Log-Einträge, sondern zur Speicherung eines Zeigers auf den aktuellen Log-Satz benutzt. Ebenso ist hier der Wert von `maxrec` dauerhaft gespeichert. Dazu werden die beiden Zeitfelder `stzeit` und `ezeit` "zweckentfremdet".

Wichtig ist, daß Sie in Ihre Anwendungsprogramme den Aufruf der Prozedur `logein` *vor* dem Öffnen jeglicher Dateien einfügen und daß Sie `logaus` aufrufen, *nachdem alle* Dateien geschlossen wurden.

Bei jedem Start der Anwendung erfolgt zunächst ein Eintrag des Start-Datums und der -Zeit in die Log-Datei, die danach sofort wieder geschlossen wird.

Wenn das Programm ordnungsgemäß beendet wird, erfolgt am Ende in denselben Satz der Log-Datei ein Eintrag von End-Datum und Zeit. Wird dagegen das Programm "unsauber" abgebrochen, erfolgt dieser Eintrag nicht.

Mit einem Datenbank-Editor (z.B. dem Programm BROWSE aus dem nächsten Kapitel) können Sie leicht herausfinden, wo die Ende-Eintragungen fehlen.

Wenn Sie in Ihrer Anwendung den Bedienernamen abfragen, können Sie die Prozeduren so erweitern, daß auch dieser in der Log-Datei "verewigt" wird.

Mit dem folgenden "Simpel-Programm" können Sie die Prozeduren testen:

```
*
* LOGTEST.PRG
*
CLEAR
DO logein
? "Anmeldung erfolgt"
INKEY(0)
DO logaus
? "Abmeldung erfolgt"
QUIT
```

Starten Sie dies Programm mehrmals und brechen Sie es manchmal mit `Alt-C` ab, nachdem die Meldung "`Anmeldung erfolgt`" auf dem Bildschirm erscheint. (Sie können auch den Rechner neu starten oder ihn aus- und einschalten.)

Wenn Sie sich danach die Log-Datei ansehen, werden Sie erkennen, wann Sie "gesündigt" haben.

19.4 "Count-Down" Anzeige bei Transaktionen

Nichts ist für den Anwender eines Programms spannender, wenn keine Reaktion auf dem Bildschirm erscheint und das Programm anscheinend nichts tut - "es wird sich doch wohl nicht aufgehangen haben?!"

Unfreiwillige Wartezeiten entstehen in einem Clipper-Programm besonders bei Stapelprozessen, wie z.B. Drucken von Listen, Reorganisieren von Datenbeständen und natürlich besonders beim Aufbau von Indexdateien.

Durch wenige Programmzeilen kann man dem Anwender zeigen, daß der Computer noch "lebt", wie folgendes Beispiel demonstriert:

```
SET CURSOR OFF
GO TOP
@ 20,5 SAY "Noch zu bearbeiten:      Sätze"
maxrec = LASTREC()
DO WHILE !EOF()
  @ 20,25 SAY maxrec PICTURE "99999"
  *
  * hier Transaktions-Anweisungen einfügen
  *
  maxrec = maxrec - 1
ENDDO
SET CURSOR ON
*
* usw.
*
```

Dieses Verfahren ist immer dann geeignet, wenn die Bearbeitung von Sätzen einer Datei unter Programmkontrolle stattfindet. Es versagt jedoch, wenn die Verarbeitung Clipper-intern erfolgt, wie z.B. bei der Anweisung INDEX...TO. Hier hilft folgender Trick:

```
PUBLIC disp
disp = .F.
*
* diverse Anweisungen
*
USE kunden
maxrec = LASTREC()
disp = .T.
```

```
@ 20,5 SAY "Noch zu bearbeiten:      Sätze"
SET CURSOR OFF
INDEX ON cnt(UPPER(name)) TO namen
disp = .F.
@ 20,5 SAY SPACE(30)
SET CURSOR ON
*
* weitere Anweisungen
*

FUNCTION cnt
PARAMETERS val
IF disp
  @ 20,25 SAY maxrec-RECNO() PICTURE "99999"
ENDIF
RETURN val
```

Bei der Bildung der Indexdatei wird die UDF `cnt()` angegeben. In dieser UDF wird die Count-Down Anzeige immer dann durchgeführt, wenn die PUBLIC-Variable `disp` den Wert `.T.` hat. Wie Sie dem Beispiel entnehmen können, wird diese Variable unmittelbar vor dem Indizieren auf `.T.` gesetzt und anschließend erhält sie wieder den Wert `.F.`

Wenn Sie Indexdateien mit der Funktion `cnt()` bilden, ist es natürlich erforderlich, daß in allen Programmen die Funktion zur Verfügung steht, die die Indexdateien benutzten, selbst dann, wenn keine Count-Down Anzeige erfolgen soll. Ebenso muß die Variable `disp` definiert sein.

Wie Tests gezeigt haben, wird der "normale" Gebrauch von Indexdateien durch dieses Verfahren kaum meßbar verlangsamt und bei der Bildung von Indexdateien mit Count-Down Anzeige ergibt sich ein Geschwindigkeitsverlust von ca. 10 bis 20%, bedingt durch die Bildschirm-Anzeige und die Subtraktion.

Allgemein sind Count-Down Anzeigen sinnvoller, als einfaches Aufwärtszählen von Sätzen, da der Anwender damit abschätzen kann, wie lange es noch dauern wird, bis Null erreicht wird, während er beim Aufwärtszählen ja oft nicht wissen kann, welches der größte Wert ist.

Besonders "elegant" sind natürlich Anzeigen wie

```
x % Daten bearbeitet
```

allerdings erfordert die Bildung von Prozentwerten eine Division, die deutlich mehr Zeit benötigt, als eine einfache Subtraktion.

19.5 "Anti-PACK" Routinen

Wie Sie wissen, werden durch die DELETE-Anweisung Sätze in einer Datei lediglich als gelöscht markiert, jedoch nicht physisch aus der Datei entfernt. Die "Leichen" werden erst durch die PACK-Anweisung tatsächlich gelöscht. Obwohl die Anweisung PACK recht simpel erscheint, veranlaßt sie doch umfangreiche Aktionen, die entsprechend viel Zeilt kosten:

1. aktuelle Datei in eine temporäre Datei kopieren und dabei als gelöscht markierte Sätze übergehen

2. aktuelle Datei löschen

3. temporäre Datei mit dem Namen der ursprünglichen Datei versehen.

Wenn Sie indizierte Dateien verwenden (und das ist ja meistens der Fall, können Sie die PACK-Anweisung getrost aus Ihrem Vokabular streichen, wenn Sie folgende Routinen ähnlich den folgenden benutzen:

```
USE kunden INDEX kdname
*
* verschiedene Anweisungen im Programmteil "Löschen"
*
* DELETE - wird nicht mehr benötigt
* PACK   - ist auch nicht mehr nötig

REPLACE name with SPACE(20)    && ist der Ersatz für DELETE

*
* Anfügen eines neuen Satzes
*
GO TOP
IF !EMPTY(name)
  APPEND BLANK
ENDIF
*
* Anweisungen zum Eintragen der Feld-Informationen
*
```

Beim Löschen eines Satzes wird anstelle der DELETE- und ggf. der PACK-Anweisung das Indexfeld des zu löschenden Satzes einfach mit einem Leerstring überschrieben. Da die Datei auf dieses Feld indiziert ist, "rutscht" dieser Satz an den Anfang der Ordnungsfolge.

Beim späteren Anfügen eines neuen Satzes wird durch die GO TOP-Anweisung zunächst ein Sprung auf den (indizierten) Anfang der Datei ausgeführt und geprüft, ob in diesem Satz das Schlüsselfeld leer ist. Falls ja, handelt es sich um einen zuvor gelöschten Satz und die Datei bleibt auf diesem Satz positioniert.

Ist am "Anfang" der Datei kein Satz mit einem leeren Schlüsselfeld vorhanden, wird mit APPEND BLANK ein neuer Satz angefügt und die Datei ist dadurch automatisch auf desen Satz positioniert.

Es sollen natürlich auch die Nachteile dieses Verfahrens nicht verschwiegen werden. Im Gegensatz zur DELETE-Anweisung besteht hier nicht die einfache Möglichkeit, mit RECALL einen gelöschten Satz wieder zu aktivieren.

Weiterhin dürfen keine "echten" Sätze in der Datei vorhanden sein, die ein leeres Schlüsselfeld enthalten, da in diesem Fall nicht zwischen tatsächlichen und gelöschen Sätzen unterschieden werden kann. (Solche Fälle sind in der Praxis jedoch sicher sehr selten.)

Wenn man einen zuvor bereits benutzten Satz erneut verwendet, müssen alle Felder mit neuen Werten gefüllt werden. Im Gegensatz zu APPEND BLANK sind diese Felder nicht in jedem Fall leer, sondern es besteht die Möglichkeit, daß noch "alte" Daten enthalten sind.

19.6 "Verzeigerte" Dateien

In einigen Fällen ist die feste Satzstruktur von Clipper und dBase zur Speicherung von Informationen mit variablem Umfang ungünstig. Denken Sie z.B. an den Fall, daß Rechnungen gespeichert werden sollen. Neben den immer gleichen Informationen, wie Anschrift des Kunden, Rechnungsnummer, Zahlungsmodalitäten und Skonti sowie Gesamtsummen müssen auch die einzelnen Posten einer Rechnung gespeichert werden.

Hier besteht nun die Problematik, daß auf einer Rechnung nur ein einziger Posten, auf einer anderen jedoch vielleicht einige hundert Posten enthalten sein können.

Das Problen so zu lösen, daß man für jede Rechnung einen Satz in einer Datei "spendiert", in dem für die größtmögliche Anzahl von Rechnungsposten Felder vorgesehen sind, ist meist nicht praktikabel, da viel Platz verschenkt wird, denn es müssen ja soviele Felder bereitgehalten werden, wie die maximale Anzahl von Posten ist. Dabei müssen für jeden Posten mehrere Felder bereitgestellt werden, wie z.B. Stückzahl und Artikelnummer.

Noch schwieriger und unhandlicher wird die Angelegenheit, wenn es nicht möglich ist, die Bezeichnung und Preise der Posten aus einer Artikeldatei auszulesen, weil vielleicht beliebige Rechnungsposten mit freiem Text und freien Preisen bei der Rechungserfassung eingegeben werden sollen.

Eine mögliche Lösung des Problems ist, daß man eine getrennte Postendatei verwaltet, in der für jeden Rechnungsposten jeder Rechnung ein Satz enthalten ist, der neben den speziellen Informationen, die sich auf den Posten beziehen (Text, Stückzahl, Preis usw.), noch ein Schlüsselfeld enthält, das eine Referenz-Nummer speichert (z.B. die Nummer der Rechnung, zu der der Posten gehört).

Zusätzlich muß eine Index-Datei auf dieses Schlüsselfeld gebildet werden, damit bei der Rechnungserstellung der erste zur Rechnung gehörige Posten schnell gefunden werden kann. Auf weitere Posten wird dann mit der SKIP-Anweisung positioniert, solange bis kein Posten mit gleicher Schlüsselnummer mehr gefunden wird.

Zusätzlicher Programmieraufwand entsteht bei dieser Lösung dadurch, daß die nicht mehr benötigten Posten in dieser Datei zum geeigneten Zeitpunkt gelöscht werden müssen, damit die Datei nicht unendlich anwächst. Da zusätzlich ein PACK erforderlich ist, wirkt sich dieses Verfahren negativ auf das Zeitverhalten der Anwendung aus.

Den folgenden Routinen liegt ebenfalls die Idee einer getrennten Posten-Datei zugrunde, die jedoch ganz anders verwaltet wird.

Die Grundstruktur dieser Datei kann z.B. so aussehen:

```
Struktur von POSTEN.DBF

Feld-Name Feld-Typ Feld-Länge Feld-Dez.
--------- -------- ---------- ---------
TEXT         C          70         0
POINTER      N           5         0
```

Natürlich können beliebige weitere Felder hinzugefügt werden. Der Schlüssel zur Satz-Verwaltung ist das Feld POINTER, über das mehrere Sätze verkettet werden.

Dabei gilt:

POINTER = 0: kein Folgesatz
POINTER <> 0: POINTER enthält die Nummer des Folgesatzes

Wenn man nun die Datei POSTEN auf den ersten Satz einer Kette positioniert (dieser muß natürlich bekannt sein), dann kann man sich von dort über das Feld POINTER durch die Kette "hangeln", bis man einen Satz erreicht, in dem das POINTER-Feld den Wert 0 enthält:

```
GO START_SATZ
DO WHILE pointer <> 0
  GO pointer
ENDDO
```

Dieses einfache Verfahren, eine Kette zu lesen ist außerdem recht schnell und es wird keine Index-Datei benötigt.

"Trickreicher" ist jedoch das Einfügen von Einträgen in die Datei, besonders da berücksichtigt werden muß, daß bestimmte Satzketten in der Datei gelöscht sein können. Da alle POINTER-Informationen über absolute Satznummern gehen, kann man nicht einfach Sätze als gelöscht markieren und diese mit PACK entfernen, da dann alle Bezüge auf Satznummern nicht mehr korrekt wären.

Gelöschte Satzketten sollten jedoch im "Recycling" wieder für neue Eintragungen zur Verfügung stehen, damit die Datei nicht ständig wächst. Bei dem hier gezeigten Verfahren nimmt dazu der erste Satz der Datei eine Sonderstellung ein. Er speichert keine echten Daten, sondern sein POINTER-Feld hat folgende Bedeutung:

POINTER = 0:	kein gelöschter Satz vorhanden
POINTER <> 0:	POINTER enthält die Nummer des ersten Satzes einer gelöschten Kette

Mit diesen Informationen kann man nun beim Einfügen neuer Informationen zunächst prüfen, ob eine gelöschte Kette vorhanden ist und diese Kette weiterverwenden. Es ist natürlich in den wenigsten Fällen so, daß die neue Information genausoviele Sätze benötigt, wie die zuvor gelöschte. Aus diesem Grund muß bei einer kleineren Anzahl neuer Sätze der POINTER im ersten Satz auf den ersten verbleibenden Satz der alten Kette gesetzt werden.

Ist die neue Kette länger als die alte gelöschte Kette, dann werden zunächst alle Sätze der alten Kette "verbraucht" und anschließend neue Sätze an die Datei angefügt. In diesem Fall muß abschließend POINTER in Satz 1 auf 0 gesetzt werden, da ja keine gelöschten Sätze mehr für Neueintragungen bereitstehen.

Das Löschen von Ketten scheint zunächst recht einfach. Die erste Überlegung ist, in Satz 1 den POINTER einfach auf den Anfang der zu löschenden Kette zeigen zu lassen. Dieses Verfahren versagt jedoch, wenn hintereinander mehrere Ketten gelöscht werden sollen. Wird dabei der POINTER immer auf den Anfang der letzten zu löschenden Kette gesetzt, so verbleiben die zuvor gelöschten Ketten als "Leichen" in der Datei.

Hier kann man dadurch Abhilfe schaffen, indem man sich dann, wenn bereits eine gelöschte Kette vorhanden ist, an das Ende dieser "hangelt" und dort den Inhalt von POINTER (der ja 0 ist) auf den Anfangssatz der neuen zu löschenden Kette setzt. Dadurch hat man einfach eine einzige, längere gelöschte Kette geschaffen.

Genug der Theorie, hier die erforderlichen Funktionen. Dabei wird davon ausgegangen, daß eine Kette aus maximal 20 Elementen besteht und daß zu schreibende und zu lesende Informationen in einem Array `txt[]` gespeichert werden. Sie brauchen natürlich nur die Arrays mit anderem Umfang zu deklarieren, um eine andere Maximalzahl zu erreichen.

```
* Schreiben einer neuen Kette in die Datei POSTEN
*
* Parameter: anzahl := Anzahl der zu schreibenden Sätze
*
* Rückgabe:  Nummer des ersten Satzes der Kette; dieser
*            Wert wird zum späteren Ansprechen der Kette
*            benötigt!
*
* Anmerkung: txt[] muß die zu schreibenden Daten enthalten.
*            Die Datei POSTEN muß vor dem Aufruf selektiert
*            sein.
*
FUNCTION putrec
PARAMETERS anzahl
PRIVATE ptr, retptr, lastptr, i

* Behandlung des Sonderfalls, daß POSTEN noch keine Sätze
* enthält: Anfügen des Verweis-Satzes für gelöschte
*          Ketten
IF LASTREC() = 0
  APPEND BLANK
  REPLACE pointer WITH 0              && keine gelöschte Kette
ENDIF

* Zeiger auf gelöschte Kette lesen, falls nicht vorhanden,
* neuen Satz anfügen.
* Auf den ersten Satz der gelöschten Kette bzw. auf den
* neuen Satz positionieren und Satz-Nummer für die Rückgabe
* speichern
GO TOP
IF pointer = 0                        && wenn keine gelöschte
*                                        Kette, dann
  APPEND BLANK                        && neuen Satz anfügen
  retptr = RECNO()                    && und für Rückgabe
*                                        speichern
ELSE                                  && sonst den Anfang der
*                                        gelöschten Kette für
  retptr = pointer                    && Rückgabe speichern
  GO pointer                          && auf Anfangssatz der
*                                        zu schreibenden Kette
*                                        positionieren
ENDIF
```

```
* Informationen schreiben
i = 1                                  && Satzzähler
DO WHILE .T.
  REPLACE text WITH txt[i]             && Information schreiben
  i = i + 1                            && Zähler erhöhen
  IF i > anzahl                        && alle Sätze ?
    EXIT                               && fertig, "Ausstieg"
  ENDIF
  IF pointer <> 0                      && weiterer gelöschter
*                                         Satz vorhanden ?
    GO pointer                         && ja, dorthin positio-
*                                         nieren
  ELSE                                 && sonst
    lastptr = recno()                  && aktuellen Satz merken
    APPEND BLANK                       && neuen Satz anfügen
    REPLACE pointer WITH 0             && "vorläufiges" Ende der
*                                         Kette markieren
    ptr = RECNO()                      && aktuellen Satz merken
    GO lastptr                         && auf vorigen Satz gehen
*                                         und dort den Verweis
*                                         auf den aktuellen Satz
    REPLACE pointer WITH ptr           && eintragen
    GO ptr                             && wieder auf aktuellen
*                                         Satz positionieren
  ENDIF
ENDDO

* Ketten-Ende markieren und Satz 1 aktualisieren
ptr = pointer                          && falls in gelöschte
*                                         Kette geschrieben wur-
*                                         de, "Fortsetzung"
*                                         merken (oder 0, wenn
*                                         neue Kette länger als
*                                         die gelöschte Kette
REPLACE pointer with 0                 && endgültiges Ende mar-
*                                         kieren
GO TOP                                 && Satz 1 aktualisieren
REPLACE pointer with ptr               && Verweis auf "Rest" der
*                                         gelöschten Kette oder
*                                         0 eintragen
RETURN retptr                          && Rückgabe: erster Satz
*                                         der neuen Kette
```

```
*
* Löschen einer Kette in der Datei POSTEN
*
* Parameter: rn := Nummer des ersten Satzes der Kette
*
* Rückgabe:  rn
*
* Anmerkung: Die Datei POSTEN muß vor dem Aufruf selektiert
*            sein.
*
FUNCTION delrec
PARAMETERS rn
GO TOP                                  && auf Satz 1
IF pointer = 0                          && keine gelöschte Kette
*                                          vorhanden
  REPLACE pointer WITH rn               && also Verweis auf ge-
*                                          löschte Kette ein-
*                                          tragen
ELSE                                    && sonst
  GO pointer                            && auf Anfang der ge-
*                                          löschten Kette po-
*                                          sitionieren
  DO WHILE pointer <> 0                 && bis an das Ende
    GO pointer                          && "durchhangeln"
  ENDDO
  REPLACE pointer with rn               && hier Anfang der neuen
*                                          zu löschenden Kette
*                                          eintragen
ENDIF
RETURN rn
```

```
*
* Lesen einer Kette aus der Datei POSTEN
*
* Parameter: rn := Nummer des ersten Satzes der Kette
*
* Rückgabe:  Anzahl der gelesenen Sätze
*
* Anmerkung: txt[] muß die zu lesenden Daten aufnehmen.
*            Die Datei POSTEN muß vor dem Aufruf selektiert
*            sein.
*
FUNCTION getrec
PARAMETERS rn
PRIVATE i

i = 1                                  && Satzzähler
GO rn                                  && auf den Anfang der
*                                         Kette positionieren
DO WHILE pointer <> 0                  && Bis zum Ende der
*                                         Kette "durchhangeln"
  txt[i] = text                        && Information lesen
  GO pointer                           && auf nächsten Satz
  i = i + 1                            && Zähler erhöhen
ENDDO
txt[i] = text                          && Inhalt des letzten
*                                         Satzes lesen
RETURN i                               && Satzzähler zurück-
*                                         geben
```

Zum praktischen Einsatz der Funktionen müssen Sie ggf. die Struktur der Datei POSTEN so ändern, daß sie die von Ihnen benötigten Felder enthält (das Feld POINTER muß natürlich erhalten bleiben). In den Funktionen getrec und putrec müssen Sie die Schreib- und Lese-Anweisungen ebenfalls anpassen und im Hauptprogramm entsprechende Arrays deklarieren.

Wenn wir auf das anfängliche Beispiel der Rechnungs-Datei zurückkommen, so muß in dieser Datei ebenfalls ein Feld aufgenommen werden, daß den Zeiger auf die entsprechende Kette in der Datei POSTEN speichert.

Bei jedem Erfassen einer neuen Rechnung lesen Sie die Informationen für die Posten in entsprechende Arrays ein, rufen danach die Funktion `putrec()` auf und speichern die Rückgabe der Funktion (den Anfang der Kette) in das Zeigerfeld der Haupt-Rechnungsdatei.

Neben der Anwendung für eine Rechnungsdatei gibt es viele weitere Möglichkeiten, "verzeigerte" Dateien zu verwenden und zwar besonders dann, wenn bestimmte Informationen zu gruppieren sind, wenn jedoch der Umfang der Informationen je Gruppe unterschiedlich groß ist. Das Verfahren wurde z.B. auch erfolgreich bei einem Programm zur Bearbeitung von Bau-Ausschreibungen benutzt. Speziell bei einer solchen Anwendung gibt es die unterschiedlichste Anzahl von Unterpunkten pro Position, so daß eine platzsparende Speicherung auf anderem Wege sehr schwierig ist.

Die folgenden Abbildungen zeigen, wie sich verschiedene Operationen auf den Inhalt der Datei POSTEN auswirken:

Leere Datei mit Satz 1

RECNO()	TEXT	POINTER
1		0

Eintrag einer Kette mit 10 Sätzen: `putrec(10) = 2`

RECNO()	TEXT	POINTER
1		0
2	Kette 1.1	3
3	Kette 1.2	4
4	Kette 1.3	5
5	Kette 1.4	6
6	Kette 1.5	7
7	Kette 1.6	8
8	Kette 1.7	9
9	Kette 1.8	10
10	Kette 1.9	11
11	Kette 1.10	0

Eintrag einer Kette mit 3 Sätzen: `putrec(3) = 12`

RECNO()	TEXT	POINTER
1		0
2	Kette 1.1	3
3	Kette 1.2	4
4	Kette 1.3	5
5	Kette 1.5	6
6	Kette 1.6	7
7	Kette 1.4	8
8	Kette 1.7	9
9	Kette 1.8	10
10	Kette 1.9	11
11	Kette 1.10	0
12	**Kette 2.1**	**13**
13	**Kette 2.2**	**14**
14	**Kette 2.3**	**0**

Löschen der Kette 1
`delrec(2)`

RECNO()	TEXT	POINTER
1		**2**
2	Kette 1.1	3
3	Kette 1.2	4
4	Kette 1.3	5
5	Kette 1.4	6
6	Kette 1.5	7
7	Kette 1.6	8
8	Kette 1.7	9
9	Kette 1.8	10
10	Kette 1.9	11
11	Kette 1.10	0
12	Kette 2.1	13
13	Kette 2.2	14
14	Kette 2.3	0

Eintrag einer Kette mit 4 Sätzen: `putrec(4) = 2`

RECNO()	TEXT	POINTER
1		**6**
2	**Kette 3.1**	3
3	**Kette 3.2**	4
4	**Kette 3.3**	5
5	**Kette 3.4**	**0**
6	Kette 1.5	7
7	Kette 1.6	8
8	Kette 1.7	9
9	Kette 1.8	10
10	Kette 1.9	11
11	Kette 1.10	0
12	Kette 2.1	13
13	Kette 2.2	14
14	Kette 2.3	0

Eintrag einer Kette mit 2 Sätzen: `putrec(2) = 6`

RECNO()	TEXT	POINTER
1		**8**
2	Kette 3.1	3
3	Kette 3.2	4
4	Kette 3.3	5
5	Kette 3.4	0
6	**Kette 4.1**	7
7	**Kette 4.2**	**0**
8	Kette 1.7	9
9	Kette 1.8	10
10	Kette 1.9	11
11	Kette 1.10	0
12	Kette 2.1	13
13	Kette 2.2	14
14	Kette 2.3	0

Löschen der Kette 4
`delrec(6)`

RECNO()	TEXT	POINTER
1		**8**
2	Kette 3.1	3
3	Kette 3.2	4
4	Kette 3.3	5
5	Kette 3.4	0
6	Kette 4.1	7
7	Kette 4.2	0
8	Kette 1.7	9
9	Kette 1.8	10
10	Kette 1.9	11
11	Kette 1.10	6
12	Kette 2.1	13
13	Kette 2.2	14
14	Kette 2.3	0

Löschen der Kette 3
`delrec(2)`

RECNO()	TEXT	POINTER
1		8
2	Kette 3.1	3
3	Kette 3.2	4
4	Kette 3.3	5
5	Kette 3.4	0
6	Kette 4.1	7
7	Kette 4.2	**2**
8	Kette 1.7	9
9	Kette 1.8	10
10	Kette 1.9	11
11	Kette 1.10	6
12	Kette 2.1	13
13	Kette 2.2	14
14	Kette 2.3	0

Eintrag einer Kette mit 12 Sätzen: `putrec(12) = 8`

RECNO()	TEXT	POINTER
1		0
2	Kette 5.7	3
3	Kette 5.8	4
4	Kette 5.9	5
5	Kette 5.10	15
6	Kette 5.5	7
7	Kette 5.6	2
8	Kette 5.1	9
9	Kette 5.2	10
10	Kette 5.3	11
11	Kette 5.4	6
12	Kette 2.1	13
13	Kette 2.2	14
14	Kette 2.3	0
15	Kette 5.11	16
16	Kette 5.12	0

19.7 Die ungleichen Zwillinge - BOF() und EOF()

Die "alten Hasen" unter den dBASE III-Programmierern wissen, daß die `EOF()`-Funktion abgefragt werden sollte, nachdem eine `LOCATE`-, `FIND`- oder `SEEK`-Anweisung ausgeführt wurde, um zu prüfen, ob der gesuchte Satz wirklich gefunden wurde. Hierzu verwendet man in Clipper besser die `FOUND()`-Funktion, die übrigens auch in dBASE III plus zur Verfügung steht.

Die Funktionen `EOF()` und `BOF()` haben aber weiterhin ihre Berechtigung, wenn es darum geht, beim "Blättern" durch eine Datei festzustellen, ob das Ende bzw. der Anfang einer Datei erreicht wurde.

Dabei muß man jedoch das unterschiedliche Verhalten der Funktionen beachten:

Wenn eine Datei auf dem ersten Satz positioniert ist, also entweder auf Satz 1 oder (bei Verwendung einer Index-Datei) auf dem ersten Satz der Index-Folge, dann liefert `BOF()` den Wert `.F.` und nicht bereits `.T.`, wie man erwarten könnte. Erst wenn man versucht, die Datei mit `SKIP -1` weiter "nach vorne" zu positionieren, ergibt `BOF()` den Wert `.T.` Wichtig ist dabei, daß nach diesem Versuch die Datei weiterhin auf dem ersten Satz positioniert ist.

Anders ist es jedoch bei der `EOF()`-Funktion. Wenn man auf dem letzten Satz angelangt ist, dann ergibt `EOF()` ebenfalls `.F.` Nach einer weiteren `SKIP`-Anweisung liefert dann auch `EOF()` den Wert `.T.` Allerdings ist nun die Datei *nicht* mehr auf dem letzten Satz positioniert, sondern der Satzzeiger steht gewissermaßen "hinter" dem letzten Satz. Die dann evtl. gelesenen Werte enthalten keine gültige Information.

In Programmen, die es dem Anwender erlauben, durch eine Datei zu "blättern" sollte nach jeder `SKIP -1` Anweisung eine Prüfung auf `BOF()` erfolgen und bei Erreichen des Dateianfangs eine entsprechende Meldung ausgegeben werden.

Beim "Vorwärtsblättern" mit `SKIP` sollte die Prüfung auf `EOF()` erfolgen. Wenn `EOF()` den Wert `.T.` ergibt, muß ebenfalls eine entsprechende Meldung ausgegeben, *zusätzlich* jedoch die Anweisung `SKIP -1` eingefügt werden, damit die Datei wieder auf dem letzten Satz positioniert ist.

19.8 Bilden von Index-Ausdrücken

Wie Sie wissen, wird durch die Anweisung `INDEX ON <Schlüssel-Ausdruck> TO <Index-Datei>` eine Indexdatei erzeugt und gleichzeitig definiert, wie der Schlüssel aufgebaut werden soll. Im einfachsten Fall gibt man einfach den Namen eines Feldes an.

Eine der Stärken von dBASE und Clipper ist es jedoch, daß Schlüssel-Ausdrücke aus mehreren Feldern gebildet werden können und daß es auch möglich ist, Funktionen in einem Schlüssel-Ausdruck zu verwenden.

Speziell beim Einsatz der `TRIM()`- oder `LTRIM()`-Funktionen bei Clipper muß man jedoch einige Punkte beachten. Clipper bestimmt bei der Ausführung der `INDEX ON...`-Anweisung auch, wie lang das Index-Feld in der Index-Datei sein muß, damit die korrekte Dateistruktur angelegt werden kann. Dabei verwendet Clipper leere Datenfelder.

Eine Datenbank enthält z.B. ein 20 Zeichen langes String-Feld `NAME`. Die Anweisung

```
INDEX ON TRIM(name) TO namindx
```

bestimmt den Platzbedarf durch die Funktion

```
TRIM("                    ")
```

was zu einer Länge von 0 führt.

Mit diesem (zwar platzsparenden) Schlüssel werden Sie später nie Daten finden.

Damit der Aufbau der Index-Datei fehlerfrei funktioniert, müssen Sie folgende Anweisung geben:

```
INDEX ON TRIM(name) + SPACE(20-LEN(TRIM(name)))
```

Durch die zusätzliche `SPACE()`-Funktion wird das Feld mit der nötigen Anzahl von Leerzeichen "aufgefüllt".

Problematisch ist auch das Indizieren auf numerische Felder, besonders dann, wenn diese Felder Nachkommastellen besitzen. Durch Rundung kann nicht immer sichergestellt werden, daß bei der Bildung des Index-Eintrages dieselbe

Ziffernfolge verwendet wird, wie bei einem anschließenden Suchen, so daß nicht gewährleistet ist, daß ein Satz später auch gefunden wird.

Es ist am sichersten, wenn man *keine* numerischen Felder zur Bildung von Index-Dateien benutzt. Wandeln Sie besser vorher numerische Werte mit der `STR()`-Funktion in Zeichenketten um und bauen Sie damit den Index auf.

Ebenso sollte man keinen Index unmittelbar mit einem Datumsfeld aufbauen, da Clipper intern ein spezielles Format für Datumswerte verwendet. Verwenden Sie die Funktion `DTOS()`, um einen Datumswert in eine Zeichenkette umzuwandeln. Diese Funktion bietet außerdem den Vorteil, daß das Datum in eine Zeichenkette mit dem Format `JJMMTT` umgewandelt wird, die die chronologisch richtige Reihenfolge sicherstellt.

19.9 Eine einfache Taschenrechner-"Pop-Up"-Routine

In vielen kommerziellen Programmen werden heute sogenannte "Pop-Up"-Routinen eingesetzt, die die Bedienung eines Programms erleichtern und die das Programm vielseitiger machen.

Mit den Möglichkeiten, die Clipper bietet, kann man derartige "Features" auch in eigene Programme ohne große Mühe einbauen. Das folgende Beispiel zeigt einen einfachen Taschenrechner für die vier Grundrechenarten. Die Besonderheit ist, daß man auch Wunsch ein Rechenergebnis in den Tastaturpuffer und damit in das jeweils aktive Eingabefeld übertragen kann.

```
************************************
*                                  *
*  Datei CALC.PRG                  *
*                                  *
*  Simulation eines Taschenrechners *
*                                  *
*  von Günther Daubach,            *
*                                  *
************************************

parameters ergebn, ok, x, y
private op_alt, versch, ok

* Bildschirm sichern und Rechner anzeigen
*
* Falls nötig, Koordinaten korrigieren, daß
* Rechner vollständig auf den Bildschirm paßt
*
x = if(x > 48,48,x)
y = if(y > 11,11,y)

do rech_anz                && Rechner anzeigen
*
* Variablen initialisieren
*
op = ""                     && Operator
op_alt = ""                 && letzer Operator
versch = .t.                && Merker "Verschieben"
ergebn = 0                  && Ergebnis
ok = .t.                    && Fehler-Merker
```

```
*
* Hauptschleife - ausführen bis
* Eingabe = "X" oder "E" (Esc)
*
do while ! (op $ "XE")
  *
  * Ersten Operanden und Operator lesen
  *
  num = "0"
  do lieszahl with num,op,x,y,op # "="
  *
  * Eingaben "Ende" und "Löschen"
  * bearbeiten
  *
  if op $ "=XEC"
    if ((op = "X") .and. (op_alt # "=")) .or. (op = "=")
      ergebn = val(num)
    endif
    op_alt = op
    @ y+2,x+2 say str(ergebn,21,4)
    @ y+4,x+26 say " "
    loop
  endif
  *
  * Ersten Operanden ins Ergebnis übernehmen
  *
  ergebn = val(num)
  *
  * weiter Operanden und Operatoren lesen
  *
  do while .t.
    op_alt = op
    do lieszahl with num,op,x,y,.f.
    *
    * Eingaben "Ende" und "Löschen" bearbeiten
    *
    if op $ "XEC"
      exit
    endif
```

```
    *
    * rechnen...
    *
    ergebn = berechnen(ergebn,num,op_alt)
    *
    * Überlauf oder Division durch Null ?
    *
    ok = !(ergebn = 999999999999999999999)
    *
    * Ergebnis anzeigen
    *
    @ y+2,x+2 say str(ergebn,21,4)
    @ y+4,x+26 say " "
    *
    * Ergebnis-Taste bearbeiten
    *
    if op = "="
      op_alt = op
      exit
    endif
  enddo
enddo
*
* Merker, wenn Ergebnis ok
*
ok = ((op = "X") .and. ok)

restscreen(y,x,y+13,x+31,puffer)
return
*
* Funktion zur Berechnung des Ergebnisses
*
function berechnen
parameters ergebn,num,operator
do case
  case operator = "+"
    return(ergebn + val(num))
  case operator = "-"
    return(ergebn - val(num))
  case operator = "*"
    return(ergebn * val(num))
```

```
  case operator = "/"
    if val(num) = 0            && Fehler: Div. durch 0
      ?? chr(7)
      return(999999999999999999999)
      ok = .f.
    else
      return(ergebn / val(num))
      ok = .t.
    endif
endcase
*
* Eine Zahl in num und in Operator nach op
* einlesen
* Anzeigeposition wird durch x und y bestimmt
* Anzeige löschen, wenn cl = .T. ist
*
procedure lieszahl
parameters num,op,x,y,cl
private dez_eing, mant_lg, dez_lg
num = "0"
dez_eing = .f.
mant_lg = 1
dez_lg = 0
*
* Anzeige löschen, falls erforderlich
*
if cl
  @ y+2,x+2 say str(val(num),16,0) + "      "
  @ y+4,x+26 say " "
endif
*
* Hauptschleife für Eingabe
*
do while .t.
  ch = liestaste()
  do case
    case ch $ "+-*/=XCE"   && Operanden und Sondertasten
      op = ch
      @ y+2,x+27 say ch
      @ y+4,x+26 say " "
      exit
```

```
  case ch = "B"           && Backspace (Eingabe löschen)
    num = "0"
    mant_lg = 1
    dez_lg = 0
    dez_eing = .f.
    @ y+2,x+2 say str(val(num),16,0) + "      "
    @ y+4,x+26 say " "
  case ch = "V"           && Vorzeichen wechseln
    num = if((dez_eing .and. (dez_lg=0)), ;
          ltrim(str(-val(num),16,0)) + ".", ;
          ltrim(str(-val(num),16,dez_lg)))
  case ch = "."           && Dezimalpunkt
    if dez_eing           && schon vorhanden ?
      ?? chr(7)
    else                  && nein, übernehmen
      num = num + "."
      dez_eing = .t.      && erst Dezimalstellen
    endif
  otherwise               && Ziffern-Eingabe
    if ! dez_eing         && links vom Punkt
      if num = "0"        && erste Ziffer?
        num = ch          && ja
      else
        if mant_lg = 10   && Überlauf?
          ?? chr(7)       && "schimpfen"
        else
          num = num + ch && nein, Ziffer akzeptiert
          mant_lg = mant_lg + 1  && Ziffernzähler
        endif
      endif
    else                  && nun Eingabe von Dezimalen
      if dez_lg = 4       && Überlauf?
        ?? chr(7)         && "schimpfen"
      else                && nein, akzeptieren
        num = num + ch
        dez_lg = dez_lg + 1       && Ziffernzähler
      endif
    endif
endcase
```

```
  *
  * Zahl anzeigen
  *
  if dez_eing                && Dezimalpunkt?
    if dez_lg = 0            && nein
      @ y+2,x+2 say str(val(num),16) + ".     "
    else                     && ja, mit Dezimalstellen
      @ y+2,x+2 say str(val(num),17+ ;
                    dez_lg,dez_lg) + ;
                    space(4-dez_lg)
    endif
  else                       && kein Dezimalpunkt
    @ y+2,x+2 say str(val(num),16) + "      "
  endif
  @ y+4,x+26 say " "
enddo
return
*
* Tasteneingabe lesen
*
function liestaste
do while .t.
  *
  * Pfeiltasten abfragen, wenn Verschiebemodus aktiv
  *
  do while .t.
    c = inkey(0)
    if versch .and. ((c=5) .or. (c=24) .or. (c=19) ;
                  .or. (c=4) .or. (c=26) .or. ;
                  (c=2) .or. (c=1) .or. (c=6))
      do versch_rech with c
    else
      exit
    endif
  enddo
  ch = upper(chr(c))
  do case
    case ch $ "0123456789+-*/=VXC."   && Ziffern oder
Sondertasten
      do versch_aus                   && Verschieben aus
      return(ch)
```

```
    case c = 8                          && Back Space
      do versch_aus
      return("B")
    case ch = ","                       && Komma --> Punkt
      do versch_aus
      return(".")
    case c = 13                         && Return --> =
      do versch_aus
      return("=")
    case c = 27                         && Esc
      do versch_aus
      return("E")
    otherwise
      ?? chr(7)                         && "schimpfen"
  endcase
enddo
*
* Verschieben ausschalten
*
procedure versch_aus
if versch
  @ y+6,x+26 say "   "
  @ y+8,x+26 say "   "
  @ y+4,x+26 say " "
  versch = .f.
endif
return
```

```
*
* Rechner anzeigen
*
procedure rech_anz

  puffer = savescreen(y,x,y+13,x+31)

@ y+ 0,x say "┌──────────────────────────────────────┐"
@ y+ 1,x say "│                                      │"
@ y+ 2,x say "│                                  0   │"
@ y+ 3,x say "├──────┬──────┬──────┬──────┬──────────┤"
@ y+ 4,x say "│ Esc  │  X   │ BSp  │  C   │          │"
@ y+ 5,x say "├──────┼──────┼──────┼──────┼──────────┤"
@ y+ 6,x say "│  7   │  8   │  9   │  +   │  < >     │"
@ y+ 7,x say "├──────┼──────┼──────┼──────┼──────────┤"
@ y+ 8,x say "│  4   │  5   │  6   │  -   │  ^ v     │"
@ y+ 9,x say "├──────┼──────┼──────┼──────┼──────────┤"
@ y+10,x say "│  1   │  2   │  3   │  *   │ V=+/-    │"
@ y+11,x say "├──────┼──────┼──────┼──────┼──────────┤"
@ y+12,x say "│  0   │ . ,  │      │  /   │ =|Ret    │"
@ y+13,x say "└──────┴──────┴──────┴──────┴──────────┘"
@ y+4,x+26 say " "

return

*
* Rechner verschieben
*
procedure versch_rech
parameters c

  restscreen(y,x,y+13,x+31,puffer)

do case
  case (c = 5) .and. (y > 0)     && Aufpfeil
    y = y - 1
  case (c = 24) .and. (y < 11)   && Abpfeil
    y = y + 1
  case (c = 19) .and. (x > 0)    && Linkspfeil
    x = x - 1
  case (c = 4) .and. (x < 48)    && Rechtspfeil
    x = x + 1
```

```
  case (c = 26)                    && Ctrl-Linkspfeil
    x = 0
  case (c = 2)                     && Ctrl-Rechtspfeil
    x = 48
  case (c = 1)                     && Home
    y = 0
  case (c = 6)                     && End
    y = 11
endcase
do rech_anz                        && neu anzeigen
return
```

Zum Aufruf des Taschenrechners aus einer Anwendung definieren Sie mit folgenden Anweisungen eine Funktionstaste, z.B. F10:

```
SET KEY -9 TO rechner              && F10 ruft Taschenrechner auf
PUBLIC erg, ok_flag, c_x, ;
       c_y, pict_str
pict_str = "99999.99"              && Format für Wertübergabe vom
*                                     Taschenrechner
c_x = 0                            && Anfangsposition für den
c_y = 0                            && Taschenrechner
*
* weitere Anweisungen
*

procedure rechner
  parameters proc, line, name      && "Blindparameter"
  erg = 0                          && Variable für Ergebnis, muß
*                                     an CALC per Referenz über-
*                                     geben werden
  ok_flag = .f.                    && Merker für korrektes Ergeb-
*                                     nis und Verlassen des Rech-
*                                     ners mit "X"
  do calc with erg, ok_flag, c_x, c_y

* Wenn gültiges Ergebnis und Rechner mit "X" verlassen wurde,
* Ergebnis in den Tastaturpuffer übertragen
  if ok_flag
    keyboard(chr(25) + transform(erg,pict_str))
  endif
return
```

20 Programm-Beispiele

In diesem Kapitel sind vollständige Clipper-Programme enthalten und beschrieben. Sie dienen einerseits zur Demonstration spezieller Clipper-Funktionen und Anweisungen, andererseits können sie aber auch nützlich zur Programm-Entwicklung oder als Anwendungsprogramme verwendet werden.

20.1 Clipper-BROWSE

Ab der Version Herbst '86 wird mit Clipper die sehr nützliche Funktion DBEDIT() mitgeliefert, mit der man Datei-Inhalte ähnlich wie mit der BROWSE-Anweisung von dBASE III anzeigen kann.

```
*
* Clipper-BROWSE
* ein einfacher Datenbank-Editor
*
* von Günther Daubach
*
CLEAR SCREEN
DECLARE indx[7]                         && Array für Index-Namen
pfad = SPACE(20)                        && Variable für Pfadname
fname = SPACE(8)                        && Variable für Dateiname
FOR i = 1 TO 7
  indx[i] = SPACE(8)                    && Index-Namen vorbelegen
NEXT

center(0,"C L I P P E R   B R O W S E")
@ 1,0 SAY REPLICATE("-",80)
@ 20,0 SAY REPLICATE("-",80)
@ 21,0 SAY "Satz-Nr.:"
@ 22,0 SAY REPLICATE("-",80)
@ 23,0 SAY "<Ret> = Änd., <F5> = Lösch., " + ;
           "<F6> = Reorg, <F10> = Neuer Satz <Esc> = Abbruch"
@ 24,0 SAY CHR(27) + " " + CHR(26) + " Ctrl-" + CHR(27) + ;
           " Ctrl-" + CHR(26) + " Home End Ctrl-Home"   + ;
           " Ctrl-End PgUp PgDn Ctrl-PgUp Ctrl-PgDn"
DO WHILE .T.                            && BROWSE solange mit neu
*                                          einzugebenden Dateien
*                                          wiederholen, bis Esc
*                                          gedrückt wird.
```

```
  curs_on()                          && Datei-Namen lesen
  @ 2,0 SAY "Pfad"
*                                       Pfad lesen und sofort
*                                       mit "VALID-Trick"
*                                       setzen
  @ 2,15 GET pfad PICT "@K" VALID stpath(pfad)
*                                       Dateinamen lesen und
*                                       prüfen, ob Date im
*                                       Pfad existiert
  @ 2,42 SAY "Datei:"
  @ 2,51 GET fname PICT "@K" VALID isdb(fname)

*                                       Index-Dateinamen lesen
*                                       und prüfen, ob diese
*                                       im Pfad existieren
  @ 3,0 SAY "Indexdateien:"
  @ 3,15 GET indx[1] PICT "@K" VALID isntx(indx[1])
  @ 3,24 GET indx[2] PICT "@K" VALID isntx(indx[2])
  @ 3,33 GET indx[3] PICT "@K" VALID isntx(indx[3])
  @ 3,42 GET indx[4] PICT "@K" VALID isntx(indx[4])
  @ 3,51 GET indx[5] PICT "@K" VALID isntx(indx[5])
  @ 3,60 GET indx[6] PICT "@K" VALID isntx(indx[6])
  @ 3,69 GET indx[7] PICT "@K" VALID isntx(indx[7])
  READ
  IF LASTKEY() = 27                  && Wenn Esc, dann Ende
    EXIT
  ENDIF
  curs_off()
  indm = ""                          && Ausdruck für "SET
*                                       INDEX TO…" bilden
  IF !EMPTY(indx[1])
    indm = indx[1]
    i = 2
    DO WHILE !EMPTY(indx[i]) .and. (i < 8)
      indm = indm + ", " + indx[i]
      i = i + 1
    ENDDO
    USE &fname INDEX &indm           && und Datei mit Index-
*                                       dateien öffnen
  ELSE
    USE &fname                       && keine Index-Dateien
*                                       angegeben
```

```
  ENDIF
  nfld = FCOUNT()                        && Feldnamen einlesen
  DECLARE felder[nfld]                   && in Array felder[]
  FOR i = 1 TO nfld
    felder[i] = FIELDNAME(i)
  NEXT
  @ 21,10 SAY RECNO() PICT "9999999"
  @ 21,58 SAY "Sätze gesamt:"
  @ 21,72 SAY LASTREC() PICT "9999999"
  DBEDIT(4,0,19,79,felder,"udf")         && Datenbank edititern
ENDDO
CLEAR SCREEN
QUIT
*
* Anwender-definierte Funktion für DBEDIT()
*
FUNCTION udf
PARAMETERS modus, i
PRIVATE akt_feld, ret, r, c
  r = ROW()                              && Cursor-Position
  c = COL()                              && speichern
  akt_feld = felder[i]                   && Feld auf dem der
*                                           DBEDIT-Cursor
*                                           steht
  @ 21,10 SAY recno() PICT "9999999" && Satz-Nr. anzeigen
  IF DELETED()                           && evtl. Löschkenner
    @ 21,35 SAY "* gelöscht *"           && anzeigen
  ELSE                                   && oder
    @ 21,35 SAY SPACE(12)                && alte Anzeige ent-
  ENDIF                                  && fernen
  ret = 1
* Fall-Unterscheidung für verschiedene Tastendrücke in
* DBEDIT()
*
  DO CASE
    CASE modus = 1                       && BOF erreicht
      @ 21,20 SAY "Datei-Anfang"
    CASE modus = 2                       && EOF erreicht
      @ 21,20 SAY "Datei-Ende"
    CASE modus = 4
      @ 21,20 SAY SPACE(12)              && EOF/BOF-Anz. löschen
```

```
    DO CASE
      CASE LASTKEY() = 27              && Esc-Taste, dann
        ret = 0                        && DBEDIT() abbrechen
      CASE LASTKEY() = 13              && Return-Taste
        @ 21,35 SAY "Feld ändern"
        curs_on()
        @ r,c GET &akt_feld            && Feld ändern
        READ
        curs_off()
        @ 21,35 SAY SPACE(11)          && Anzeige löschen

*                                         Prüfen, ob Änderung
*                                         in einem Indexfeld
*                                         erfolgte, dann neue
*                                         Anzeige, damit Satz
*                                         an richtiger Position
*                                         erscheint
        IF felder[i] $ UPPER(INDEXKEY(0))
          ret = 2
        ENDIF                          && nötig
      CASE LASTKEY() = -9              && F10 ?
        APPEND BLANK                   && leeren Satz anfügen
*                                         und neue Gesamt-Satz-
*                                         zahl anzeigen
        @ 21,72 SAY LASTREC() PICT "9999999"
      CASE LASTKEY() = -4              && F5 ?
        IF !DELETED()                  && Satz löschen
          DELETE
          @ 21,35 SAY "* gelöscht *"
        ELSE                           && bzw.
          RECALL                       && aktivieren
          @ 21,35 SAY SPACE(12)
        ENDIF
      CASE LASTKEY() = -5              && F6 ?
        PACK                           && reorganisieren
*                                         neue Gesamt-Satzzahl
*                                         anzeigen
        @ 21,72 SAY LASTREC() PICT "9999999"
        ret = 2                        && neue Anzeige
      ENDCASE
    ENDCASE
RETURN ret
```

```
*
* Pfad setzen
*
FUNCTION stpath
PARAMETERS p
  IF !EMPTY(p)
    SET PATH TO &p
  ELSE
    SET PATH TO
  ENDIF
RETURN .T.
*
* Prüfen, ob DBF-Datei im aktuellen Pfad existiert
*
FUNCTION isdb
PARAMETERS N
  IF EMPTY(n)
    f_meldung(15,"Bitte Datei-Namen angeben")
    RETURN .F.
  ENDIF
  IF !FILE(n + ".DBF")
    f_meldung(15,"DBF-Datei nicht im aktuellen Pfad")
    RETURN .F.
  ELSE
    RETURN .T.
  ENDIF
*
* Prüfen, ob NTX-Datei im aktuellen Pfad existiert
*
FUNCTION isntx
PARAMETERS N
  IF EMPTY(n) .OR. FILE(n + ".NTX")
    RETURN .T.
  ELSE
    f_meldung(15,"NTX-Datei nicht im aktuellen Pfad")
    RETURN .F.
  ENDIF

*
* Die Funktionen f_meldung(), curs_off(), curs_on() und
*                center() finden Sie in Kapitel 7
*
```

Dieses Programm ist ein kleiner, nützlicher Datenbank-Editor. Im Normalfall können Sie Clipper-DBF-Dateien natürlich auch mit dBASE III editieren, allerdings nur, wenn die Sätze nicht mehr als 128 Felder enthalten und wenn Sie keine Index-Dateien verwenden wollen.

Mit Clipper-BROWSE bestehen diese Einschränkungen nicht. Wenn Sie indizierte Dateien bearbeiten, sollten Sie *alle* Index-Dateien angeben, damit diese beim Ändern von Schlüsselfeldern oder beim Löschen bzw. Anfügen von Sätzen aktualisiert werden.

Das Programm demonstriert den Einsatz der Funktion `DEBEDIT()` und zeigt, wie eine UDF aufgebaut werden kann, die von `DBEDIT()` aufgerufen wird und wie man darin auf verschiedene Tastendrücke reagiert.

Beachten Sie auch, wie bei der Eingabe eines Pfades und der Dateinamen sofort über eine mit `VALID` aktivierte UDF der aktuelle Pfad umgeschaltet wird und wie die Prüfung auf Existenz der Dateien erfolgt.

Sie sehen auch, wie mit den Funktionen `FCOUNT()` und `FIELDNAME()` die Feldnamen einer Datenbank bestimmt werden können.

Interessant ist auch, wie mit der Anweisung

```
IF felder[i] $ UPPER(INDEXKEY(0))
```

untersucht wird, ob ein Feld "Mitglied" eines Index-Ausdruckes ist. Wichtig ist, daß das Ergebnis von `INDEXKEY()` in Großschrift umgewandelt wird, da im Titelsatz der Index-Datei der Schlüsselausdruck stets in der Schreibweise gespeichert ist, wie er beim `INDEX ON <Ausdruck> TO <Index-Datei>` angegeben wurde, also evtl. in Kleinschrift. Im Gegensatz dazu liefert die Funktion `FIELDNAME()`, die benutzt wurde, um die Array-Elemente `felder[]` zu belegen, den Feldnamen stets in Großschrift zurück.

Sie werden sich vielleicht wundern, warum die Namen der Index-Dateien mit sieben einzelnen `GET`-Anweisungen gelesen werden, obwohl sich hier eine Konstruktion

```
FOR I = 1 TO 7
  @ 3,i*9 + 6 GET indx[i] PICTURE "@K" VALID isntx(indx[i])
NEXT
```

anbietet.

Wenn Sie das probieren, werden Sie einen Laufzeitfehler erhalten, der einen unerlaubten Array-Index meldet. Das rührt daher, daß die Laufvariable `i` nur beim Aufbau der Bildschirm-Maske die Werte 1 bis 7 durchläuft, nicht jedoch bei der nachfolgenden Bildschirm-Eingabe. Dadurch wird `isntx(indx[i])` mit einem nicht definierten Wert für `i` aufgerufen.

Das Programm soll Sie anregen, weitere Ergänzugen vorzunehmen, z.B. das Öffnen und Anzeigen mehrerer Dateien, die mit `SET RELATION` miteinender "gekoppelt" sind.

Auch eine Suchfunktion oder ein globales Ersetzen bestimmter Feld-Inhalte, abhängig von einer Bedingung oder die Eingabemöglichkeit bestimmter Anzeige-Filter sind sinnvolle Ergänzungen.

20.2 Clipper-CREATE

Auch dieses Programm verwendet als "Kernstück" die Funktion `DBEDIT()`. Es dient zum Anlegen neuer Dateistrukturen. Sie können mit diesem Programm auch Strukturen existierender Dateien ändern, ohne daß die Datei-Inhalte verlorengehen.

```
*
* CR - Create
*
* Anlegen und Ändern von Datenbank-Strukturen
*
* von Günther Daubach
*
PARAMETERS fnm

DECLARE felder[4]                    && Arrays für DBEDIT()
DECLARE titel[4]                     && belegen
felder[1] = "FIELD_NAME"
felder[2] = "FIELD_TYPE"
felder[3] = "FIELD_DEC"
felder[4] = "FIELD_LEN"
titel[1]  = "Name"
titel[1]  = "Typ"
titel[1]  = "Länge"
titel[1]  = "Dez."
```

```
mod = .F.                                && Merker f. Struktur-
*                                           änderung
IF PCOUNT() <> 1                         && DOS-Parameter
  ?
  ?
  ? "Aufruf: CR <Dateiname> (ohne '.DBF')"
  ?
  ?
  QUIT                                   && CR wurde nicht mit
ENDIF                                    && einem Dateinamen
*                                           aufgerufen

fnm = UPPER(fnm)                         && Dateiname groß
CLEAR SCREEN
curs_off()
@ 1,0 SAY "CREATE - Hifsprogramm"
@ 22,0 SAY REPLICATE("-",80)
@ 23,0 SAY "<Ret> = Ändern, <Esc> = Ende, " + ;
           "<Del> = Feld löschen, <Ins> = Feld einfügen"
@ 24,0 SAY CHR(27) + ", " + CHR(26) + ", PgUp, PgDn, " + ;
           "Ctrl-PgUp, Ctrl-PgDn, Home, End"
*
* wenn Datei vorhanden, dann Struktur ändern
*
IF FILE(fnm + ".DBF")
  @ 1,30 SAY "Ändern der Struktur von " + fnm + ".DBF"
  USE &fnm
  COPY TO stru STRUCTURE EXTENDED        && Strukturdatei anlegen
  CLOSE DATA
  mod = .t.                              && Merker "Änderung"
ELSE
*
* Datei nicht vorhanden, neue Struktur
*
  @ 1,30 SAY "Neuanlegen der Struktur von " + fnm + ".DBF"
  CREATE stru                            && Strukturdatei neu
*                                           erzeugen
ENDIF
USE stru
IF !mod                                  && bei Neuanlage einen
  APPEND BLANK                           && leeren Satz an die
ENDIF                                    && Strukturdatei anfügen
```

```
@ 3,21,20,58 BOX CHR(201)+CHR(205)+;
                 CHR(187)+CHR(186)+;
                 CHR(188)+CHR(205)+;
                 CHR(200)+CHR(186)
* Struktur editieren
DBEDIT(4,22,19,57,felder,"udf",0,"titel")
DELETE ALL FOR EMPTY(FIELD_NAME)      && leere Sätze entfernen
PACK
CLOSE DATA
IF mod                                && geänderte Struktur
  IF jn_meldung(20,"Änderungen übernehmen")
    @ 20,0 SAY "Neue Struktur anlegen... "
    CREATE xcrx FROM stru             && temporäre Datenbank
*                                        anlegen

    USE xcrx                          && Sätze aus Original-
    @ 20,0 SAY "Datensätze anfügen...    "
    APPEND FROM &fnm                  && datei anfügen
    CLOSE DATA

    @ 20,0 SAY "Backup-Datei anlegen...  "
    IF FILE(fnm + ".BAK")             && ggf. alte BAK-Datei
      tx = "del " + fnm + ".bak"      && löschen
      ERASE &tx
    ENDIF
    IF FILE(fnm + ".BAM"              && und auch Backup
      tx = "del " + fnm + ".bam"      && einer .DBT-Datei
      ERASE &tx                       && löschen
    ENDIF
*
* Originaldatei umbenennen .DBF -> .BAK
*
    tx = "rename " + fnm + ".DBF " + fnm + ".BAK"
    RUN &tx
*
* ggf. auch .DBT -> .BAM umbenennen
*
    IF FILE(fnm + ".dbt")
      tx = "rename " + fnm + ".dbt " + fnm + ".bam"
      RUN &tx
    ENDIF
```

```
* temporäre Datei in Originalnamen umbenennen
    @ 20,0 SAY "Neue Datenbank anlegen..."
    tx = "rename xcrx.dbf " + fnm + ".DBF"
    RUN &tx
* ggf. auch die .DBT-Datei
    IF FILE("xcrx.dbt")
      tx = "rename xcrx.dbt " + fnm + ".dbt"
      RUN &tx
    ENDIF
  ENDIF
  ERASE stru.dbf
ELSE                                && neue Struktur
  IF jn_meldung(20,"Datenbank anlegen")
    CREATE fnm FROM stru            && anlegen
  ENDIF
ENDIF
ERASE stru.dbf                      && Strukturdatei
*                                      löschen
curs_on()
CLEAR SCREEN
QUIT
*
* Anwender-definierte Funktion für DBEDIT()
*
FUNCTION udf
PARAMETERS modus, i
PRIVATE curs_feld, ret, r, c, rn
r = ROW()                           && Cursor-Position
c = COL()                           && sichern
DO CASE                             && Verteiler für
*                                      verschiedene
*                                      Tastendrücke
  CASE modus < 4                    && keine Taste von
    RETURN(1)                       && Interesse
  CASE LASTKEY() = 27               && Esc - Ende
    RETURN(0)
  CASE LASTKEY() = 13               && Ret - Feld ändern
    curs_on()
    DO CASE                         && Verteiler für die
*                                      verschiedenen Felder
```

```
      CASE i = 1                        && field_name
        @ r,c GET field_name PICT "!!!!!!!!!!!" ;
                              VALID leerp(field_name)
        READ
        ret = 1
        IF mod                          && wenn Änderung, Cursor
          next_field()                  && in das nächste Feld
        ELSE                            && sonst zusätzlich
          enter_next()                  && Return "drücken"
        ENDIF
      CASE i = 2                        && field_type
        @ r,c GET field_type PICT "!" ;
                              VALID typ_chk(field_type)
        READ
*
* Automatische Belegeung von field_len für alle Felder,
* die nicht vom Typ C oder N sind
*
        IF !(field_type $ "CN")
          DO CASE
            CASE field_type = "D"       && Datum
              REPLACE field_len WITH 8
            CASE field_type = "L"       && Logisch
              REPLACE field_len WITH 1
            CASE field_type = "M"       && Memo
              REPLACE field_len WITH 10
          ENDCASE
          REPLACE field_dec WITH 0      && field_dec hier immer 0
          IF RECNO() = LASTREC()        && wenn letzter Satz,
            APPEND BLANK                && neuer, leerer Satz
            next_page()
          ELSE
            next_home()
          ENDIF
          ret = 2
        ELSE                            && bei Typ C oder N
*                                          Cursor auf field_len
          IF mod
            next_field()
          ELSE                          && und bei Neuanlage
            enter_next()                && Return "drücken"
          ENDIF
```

```
          ret = 1
        ENDIF
      CASE i = 3                        && field_len
        @ r,c GET field_len PICT "999"
        READ
*
* bei Typ C field_dec immer mit 0 belegen
*
        IF field_type = "C"
          REPLACE field_dec WITH 0
          IF RECNO() = LASTREC()        && ggf. leeren Satz
            APPEND BLANK                && anfügen
            next_page()
          ELSE
            next_home()
          ENDIF
          ret = 2
*
* bei Typ N Cursor auf field_dec positionieren
*
        ELSE
          IF mod
            next_field()
          ELSE
            enter_next()                && bei Neuanlage
*                                          Return "drücken"
          ENDIF
          ret = 1
        ENDIF
      CASE i = 4                        && field_dec
        @ r,c GET field_dec PICT "999"
        READ
        ret = 1
        IF RECNO() = LASTREC()          && ggf. leeren Satz
          APPEND BLANK                  && anfügen
          ret = 2
          next_page()
        ELSE
          next_home()
        ENDIF
    ENDCASE
    curs_off()
```

```
    RETURN ret                          && je nach vorange-
*                                          gangener Operation
*                                          Anzeige auffrischen
*                                          (ret = 2) oder nicht
*                                          (ret = 1)
  CASE LASTKEY() = 7                    && Del-Taste
    rn = RECNO()                        && Satzzeiger merken
    DELETE                              && akt. Satz löschen
    PACK                                && und entfernen
    GO rn                               && Satzzeiger auf den nun
*                                          aufgerückten Satz
*                                          positionieren
    home()
    RETURN 2
  CASE LATKEY() = 22                    && Inst-Taste
    insbefore(RECNO())                  && Neuen Satz einfügen
    home()
    RETURN 2
  OTHERWISE                             && ungültige Taste
    RETURN 1
ENDCASE
*
* Funktion simuliert die INSERT BEFORE Anweisung von dBase III
*
FUNCTION insbefore
PARAMETERS rn                           && Satz-Nr., vor der ein-
*                                          gefügt werden soll
PRIVATE fn, ft, fl, fd
APPEND BLANK                            && "hinten" leeren Satz
*                                          anfügen und auf diesen
GO BOTTOM                               && positionieren
*
* satzweise Inhalt des vorangehenden in den aktuellen Satz
* kopieren, solange bis die "Einfügestelle" erreicht ist;
* Sonderfall "Einfügen am Dateianfang" wird berücksichtigt
*
DO WHILE (RECNO() >= rn) .AND. (RECNO() <> 1)
  SKIP -1                               && Inhalt des voran-
  fn = field_name                       && gehenden Satzes in
  ft = field_type                       && Speichervariablen
  fl = field_len                        && lesen
  fd = field_dec
```

```
  SKIP                                  && wieder auf akt. Satz
  REPLACE field_name WITH fn            && Inhalt des vorigen
  REPLACE field_type WITH ft            && Satzes aus den Spei-
  REPLACE field_len  WITH fl            && chervaiablen über-
  REPLACE field_dec  WITH fd            && tragen
  SKIP -1                               && einen Satz zurück
ENDDO
GO rn                                   && auf "eingefügten" Satz
REPLACE field_name WITH SPACE(8)        && gehen und Inhalt
REPLACE field_type WITH " "             && löschen
REPLACE field_len WITH 0
REPLACE field_dec WITH 0
RETURN .T.
*
* Prüfen, ob übergebener Parameter leer ist und ggf.
* Fehlermeldung ausgeben
*
FUNCTION leerp
PARAMETERS txt
IF !EMPTY(txt)
  RETURN .T.
ELSE
  f_meldung(20,"Feldname darf nicht leer bleiben")
  RETURN .F.
ENDIF
*
* Prüfen, ob Übergabeparameter zulässigen Typkenner enthält
* und ggf. Fehlermeldung ausgeben
*
FUNCTION typ_chk
PARAMETERS t
IF t $ "CNDLM"
  RETURN .T.
ELSE
  f_meldung(20,"Nur C, N, D, L oder M eingeben")
  RETURN .F.
ENDIF
*
* "Tastenfolge" für "Cursor rechts" erzeugen
FUNCTION next_field
KEYBOARD CHR(4)
RETURN .T.
```

```
*
* "Tastenfolge" für "Cursor rechts" und "Return" erzeugen
*
FUNCTION enter_next
KEYBOARD CHR(4) + CHR(13)
RETURN .T.
*
* "Tastenfolge" für "Cursor ab" und "Home" erzeugen
*
FUNCTION next_home
KEYBOARD CHR(24) + CHR(1)
RETURN .T.
*
* "Tastenfolge" für "Home" erzeugen
*
FUNCTION home
KEYBOARD CHR(1)
RETURN .T.
*
* "Tastenfolge" für "Home"  und "Crel-PgDn erzeugen
*
FUNCTION home
KEYBOARD CHR(1) + CHR(30)
RETURN .T.
```

Dieses Programm demonstriert, wie man ein Hilfsprogramm zum Anlegen neuer und zum Ändern vorhandener Datenbank-Strukturen schreiben kann. Beachten Sie beim Ändern existierender Strukturen, daß Informationen aus der "alten" Datenbank nur in solche Felder übernommen werden, deren Name nicth geändert wurde. Sie sollten bei diesen Feldern auch den Typ nicht ändern.

Als Sicherheitsfaktor wurde eine automatische Sicherung der Originaldateien eingebaut, die auch die .DBT-Dateien berücksichtigt. Die Original .DBF-Datei erhält den Zusatz .BAK und die .DBT-Datei wird mit dem Zusatz .BAM gespeichert.

Wie bereits im Programm BROWSE wird auch hier häufig von der KEYBOARD-Anweisung Gebrauch gemacht, um innderhalb der UDF für DBEDIT Tastendrücke zu simulieren.

Beim Neuanlegen einer Struktur "drückt" das Programm an einigen Stellen selbständig die Return-Taste, um die Eingabe in ein Feld zu aktivieren, wenn dies folgerichtig ohnehin nötig ist.

So wird z.B. nach der Eingabe eines Feldnamens der Cursor automatisch in das Feld `field_type` gebracht und `Return` "gedrückt", da bei einem neu anzulegenden Feld die Angabe eines Typs zwingend erforderlich ist.

Sie sehen auch, wie bei bestimmten Feld-Typen die Längen- und Dezimal-Felder automatisch übersprungen und automatisch mit den Standardwerten vorbelegt werden.

Die im Programm verwendete Funktion `insbefore()` simuliert die dBASE-Anweisung `INSERT BEFORE`, die bei Clipper micht zur Verfügung steht. Bei "normalen" Anwendungen kann man auf diese Anweisung stets verzichten, da man durch eine Index-Datei das "Einfügen" an beliebiger Stelle einer Datenbank erreichen kann. Außerdem ist die Verwendung einer INdex-Datei (auch bei dBASE) erheblich schneller, als `INSERT BEFORE`.

In dieser speziellen Anwendung, bei der keine sehr umfangreichen Datenbanken benutzt werden, ist das "Umschaufeln" der einzelnen Sätze zeitlich vertretbar. Benutzen Sie dieses Verfahren jedoch besser nicht bei einer Datenbank mit mehreren tausend Sätzen... außer Sie möchten eine verlängerte Mittagspause haben.

21 Rückgabecodes von INKEY() und LASTKEY()

Die nachstehende Tabelle enthält die Codes, die von den Funktionen `INKEY()` und `LASTKEY()` zurückgegeben werden. Teilweise können (besonders bei den Steuertasten) zwischen den verschiedenen Computersystemen Unterschiede bestehen, die vom benutzten Tastaturtreiber abhängt.

Die hier angegebenen Codes können Sie auch zusammen mit dem `KEYBOARD`-Kommando verwenden, um einen bestimmten Tastendruck zu simulieren. So bildet z.B. das Kommando

```
KEYBOARD CHR(13)
```

das Drücken der `<Return>`-Taste nach.

Taste	Code	Bemerkung	Taste	Code	Bemerkung
A	65		a	97	
B	66		b	98	
C	67		c	99	
D	68		d	100	
E	69		e	101	
F	70		f	102	
G	71		g	103	
H	72		h	104	
I	73		i	105	
J	74		j	106	
K	75		k	107	
L	76		l	108	
M	77		m	109	
N	78		n	110	
O	79		o	111	
P	80		p	112	
Q	81		q	113	
R	82		r	114	
S	83		s	115	
T	84		t	116	
U	85		u	117	
V	86		v	118	
W	87		w	119	
X	88		x	120	
Y	89		y	121	
Z	90		z	122	

Taste	Code	Bemerkung
Ctrl-A	1	
Ctrl-B	2	
Ctrl-C	3	
Ctrl-D	4	
Ctrl-E	5	
Ctrl-F	6	
Ctrl-G	7	
Ctrl-H	8	
Ctrl-I	9	
Ctrl-J	10	
Ctrl-K	11	
Ctrl-L	12	
Ctrl-M	13	
Ctrl-N	14	
Ctrl-O	15	
Ctrl-P	16	
Ctrl-Q	17	
Ctrl-R	18	
Ctrl-S		Anhalten der Bildschirm-Anzeige
Ctrl-T	20	
Ctrl-U	21	entspr. §
Ctrl-V	22	
Ctrl-W	23	
Ctrl-X	24	
Ctrl-Y	25	
Ctrl-Z	26	
Alt-A	286	
Alt-B	304	
Alt-C		Unterbrechung des Programms
Alt-D		Aufruf des Debuggers
Alt-E	274	
Alt-F	289	
Alt-G	290	
Alt-H	291	
Alt-I	279	
Alt-J	292	
Alt-K	293	
Alt-L	294	
Alt-M	306	
Alt-N	305	
Alt-O	280	
Alt-P	281	
Alt-Q	272	
Alt-R	275	

Taste	Code	Bemerkung
Alt-S	287	
Alt-T	276	
Alt-U	278	
Alt-V	303	
Alt-W	273	
Alt-X	301	
Alt-Y	277	
Alt-Z	300	
0	48	
1	49	
2	50	
3	51	
4	52	
5	53	
6	54	
7	55	
8	56	
9	57	
Alt-0	385	
Alt-1	376	
Alt-2	377	
Alt-3	378	
Alt-4	379	
Alt-5	380	
Alt-6	381	
Alt-7	382	
Alt-8	383	
Alt-9	384	
§	21	Entspr. Ctrl-U
Leer	32	
!	33	
"	34	
#	35	
%	37	
&	38	
´	39	
(	40	
)	41	
*	42	
+	43,	
,	44	
-	45	
.	46	
/	47	
:	58	
;	59	
<	60	

Taste	Code	Bemerkung
=	61	
>	62	
?	63	
@	64	
[	91	
\	92	
]	93	
^	94	
_	95	
`	96	
{	123	
|	124	
}	125	
~	126	
F1	28	Aufruf von HELP
F2	-1	
F3	-2	
F4	-3	
F5	-4	
F6	-5	
F7	-6	
F8	-7	
F9	-8	
F10	-9	
Sh-F1	-10	
Sh-F2	-11	
Sh-F3	-12	
Sh-F4	-13	
Sh-F5	-14	
Sh-F6	-15	
Sh-F7	-16	
Sh-F8	-17	
Sh-F9	-18	
Sh-F10	-19	
Ctl-F1	-20	
Ctl-F2	-21	
Ctl-F3	-22	
Ctl-F4	-23	
Ctl-F5	-24	
Ctl-F6	-25	
Ctl-F7	-26	
Ctl-F8	-27	
Ctl-F9	-28	
Ctl-F10	-29	

Taste	Code	Bemerkung
Alt-F1	-30	
Alt-F2	-31	
Alt-F3	-32	
Alt-F4	-33	
Alt-F5	-34	
Alt-F6	-35	
Alt-F7	-36	
Alt-F8	-37	
Alt-F9	-38	
Alt-F10	-39	
Home	1	
Ctl-Home	29	
PgDn	3	
Ctl-PgDn	30	
Rechts-Pfeil	4	
Ctl-Rechts-pfeil	2	
Aufpfeil	5	
Ctl-Aufpfeil		meistens ohne Rückgabe
End	6	
Ctl-End	23	
Del	7	
BackSp	8	
Tab	9	
Sh-Tab	271	
Return	13	
PgUp	18	
Ctl-PgUp	31	
Links-pfeil	19	
Ins	22	
Abpfeil	24	
Ctl-Abpfeil		meistens ohne Rückgabe
Ctl-Links-pfeil	26	
Esc	27	

22 Warenzeichen

Im Buch wurden folgende Produktnamen bzw. Warenzeichen erwähnt:

Produktname	Inhaber des Warenzeichens
Clipper	Nantucket
dBase II	Ashton Tate
dBase III	Ashton Tate
ETP-Text- und Programm-Editor	KRS Unternehmensberatung-EDV GmbH
Grafik-Toolbox	KRS Unternehmensberatung-EDV GmbH
KRS-Help	KRS Unternehmensberatung-EDV GmbH
LIB	Microsoft
LINK	Microsoft
MAKE	Microsoft
MASM86	Microsoft
MS-C	Microsoft
MS-DOS	Microsoft
Norton Editor	Peter Norton
PC-DOS	International Business Machines
PE	International Business Machines
PLINK86	Phoenix
Super-Toolbox	KRS Unternehmensberatung-EDV GmbH
TLINK	Borland International
Turbo Pascal	Borland International
WORD	Microsoft
WordStar	Micropro
1-2-3	Lotus

23 Stichworte

Abfragetext ausgeben 74
aktives Laufwerk bestimmen 113
aktuellen Pfad bestimmen 114
Antworten lesen 74
anwender-definierte Funktionen 55, 63
- in anderen Sprachen 87
- in Assemblersprache 93
- in C 143
- in Clipper Sprache 71
Arrays, mehrdimensionale 217
Assembler-UDFs, Aufruf aus Clipper-Programmen 99
-, universelle Makro-Datei für 100
-, Rahmenprogramm für 101
Aufbau eines Clipper-Programms 21

BEGINAREA 40, 46
Beispiel, Overlay Struktur 38
Beispielprogramme 253
Bibliothek-Datei 14, 16
-, eigene 214
Bildschirm-Fenster festlegen 201
- rollen/löschen 200
Bildschirm-Modus umschalten 123
BOF() 241
BROWSE.PRG 253

CALL 100
ch_at() 71
CLIPPER.LIB 199
Clipper Compiler 16
- Erweiterungen gegenüber dBase 18
- Programm, Aufbau 21
COBOL 9, 16
Compiler 14
Compilieren eines Clipper-Programms 28
- einzelner Dateien 29
Coutn-Down Anzeige 225
CREATE - CR.PRG 259
CREATE FROM 161
CURSFORM.ASM 112

Cursor ausschalten 70, 200
- positionieren 199
Cursorform ändern 112

Dateien anlegen 159
- prüfen 159
-, Kompatibilität von 17
Datenbank-Programmiersprache 12
Datum prüfen 78
dBase-Datenbanksprache 9, 10
-, Interpreter 9
-, RunTime 13
DBEDIT() 17
dec_set() 155
defa_set() 155
Definition einer Funktion 65
DO 55
DRIVE.ASM 113
Druck-Ausgabe, direkte 202
-, in Datei umleiten 218

Editor 26
Eingabeparameter 63
ENDAREA 40, 46
EOF() 241
__EXMGRAB() 91
__EXMBACK() 91
EXTEND.H 144
EXTENDA.MAC 101

Fakultät berechnen 86
Farbpalette setzen 127
Fehlermeldung anzeigen 73
Fenster aktivieren 201
FILE() 159
FILL.ASM 118
FTEST.PRG 157
FUNCTION 65
Funktion 63
-, anwender-definierte 63
-, Definition einer 65
-, private Variablen einer 66

Funktion, rekursiv 85
-, Testprogramm für 157
Funktionsauswahl 21

G_BOX.ASM 136
G_BOXF.ASM 136
G_CLS.ASM 125
G_DOT.ASM 129
G_HGR.ASM 126
G_LINE.ASM 130
G_PAL.ASM 127
G_MODE.ASM 123
G_RSCR.ASM 139
G_WSCR.ASM 137
Grafik-Funktionen 123
-, Beispiele zu 135
Grafik-Toolbox 141
Grafikschirm löschen 125
- aus einer Datei laden 139
- in eine Datei speichern 137
Grundgröße einer Programmdatei 35, 51
Gültigkeit von Variablen 55

Hauptmodul 21, 37
- bei Overlays 51
Header-Datei EXTEND.H 144
HELP.PRG 167
-, Bildschirm löschen 170
-, Dateien innerhalb von 170
-, GET...READ innerhalb von 170
- bei INKEY() 174
-, universelles Modul für 171
Hilfsmodule bei Overlays 49
Hilfsroutinen 22
Hintergrundfarbe setzen (Grafikschirm) 126

in_range() 82
Index-Datei, Schlüsselausdruck bestimmen 163
- bilden 242
interne Namen 202
Interpreter 11
INKEY() - Rückgabecodes 269

INTO 41
ischar() 107
isdrive() 116
is_altc() 153
is_altern() 153
is_bell() 153
is_century() 153
is_conf() 154
is_cons() 154
is_dele() 154
is_delim() 154
is_devpr() 155
is_esc() 155
is_exact() 154
is_exclus() 154
is_fixed() 154
is_intens() 155
is_print() 155
is_score() 155
is_soft() 155
is_wrap() 155
is_uniq() 155

Ja/Nein-Abfrage 74
jn_meldung() 74

Kompatibilität von C-Compilern 143
- von Dateien 18
KRS Text- und Programmeditor 27

Label-Dateien 22
LASTKEY() - Rückgabecodes 269
Laufwerk prüfen 116
LIB.EXE 213
Library 14
- Manager 15
Linie zeichnen, im Grafikschirm 130
LINK.EXE 32
Linker 14
Linken eines Clipper-Programms 32
Log-Datei 221

lookup() 148
LTRIM() in Index-Ausdruck 242

MAKE 29
MAKESTRU.PRG 164
Makro-Datei für Assembler-UDFs 94
marg_set() 155
MENU TO 185
- zur Satzauswahl 194
Menü 21

Nantucket 16
nchars() 150
Netzwerk-Umgebung 16
numerischer Wert, Bereichsprüfung 82

Objektdatei 14
OVERLAY.LIB 37
Overlays 35
Overlay-Bereiche 36
- Bereiche, mehrere 43
- Konzept 36
Struktur, Beispielprogramm 38
- Stuktur, Planung 48
- Technik 35
OVL-Datei 37

Parameter 25, 55
_PARCLEN() 90
_PARCSIZ() 91
_PARINFA() 90, 96
_PARINFO() 89, 95
Parameter-Prüfung, Aufrufe zur 89, 95
- Rückgabe, Aufrufe zur 89, 95
- Übergabe, Aufrufe zur 87, 97
- Liste 25, 64
PARAMETERS 56, 65
path_set() 155
PE - Personal Editor 27
Pfad bestimmen 114
PLINK86 32, 37
Pop-Up Routinen 244

Position in einer Zeichenkette abfragen 71
pot() 84
Potenzberechnung 84
_POUT 202
PRIVATE 60, 66
Programm-Sicherheit 220
- Überwachung 220
Programmdatei, Grundgröße einer 35, 51
Programmiersprache, Datenbank- 12
Prozedur-Dateien 22
-, Aufruf von 55
-, Definition von 26
-, Struktur von 56
Prozeduren 16, 55
-, Gliederung eines Programms in 25
-, Wertübergabe an 57
PUBLIC 60, 69
Pull-Down Menüs 189
Punkt setzen im Grafikschirm 129
PROMPT 187

Quellprogramm 14
-, Eingeben eines 26

rekursive Funktion 85
Report-Dateien 22
reservierte Wörter 202
RETURN 56, 66
RPG 9
Rückgabe einer Funktion 63
Rückgabewert 65
Run-Time Paket 13

SECTION 41, 46
- INTO 41, 47
_SCROLL 200
_SETCURS 139
_SETCTYP 200
SET-Parameter abfragen 152
SET PROCEDURE TO 22, 57
SET KEY...TO... 177
_SETWIN 201

SOUND.ASM 109
Steuerzeichen in Quelldateien 26
String-Variable, Prüfung, ob leer 81
strukturiertes Programmieren 16
Strukturdatei 161
Submenüs 22
Super-Toolbox 121

t_diff() 83
Taschenrechner 244
Tastendruck lesen 72
Text-Ausgabe, zentriert 84
Töne erzeugen 109
TRIM() in Index-Ausdruck 242
Turbo Pascal Editor 27

UDF 63
Untermodule 21
-, bei Overlays 49
Unterprogramm 55

VALID 181
-, englische Systemmeldungen umgehen mit 184
Variablen, Gültigkeit von 57
-, private in Funktionen 66
Variablen-Übergabe, per Referenz 59, 69
-, als Wert 53, 68
Verarbeitungsgeschwindigkeit 12
verkettete Sätze 229

Warenzeichen 273
Werte-Übergabe bei Prozeduren 57
WHATDIR.ASM 114
wild() 145
WordStar 27

Zeichenkette absuchen 148
-, bestimmte Zeichen zählen 150
-, Position abfragen 71
Zeichenkettenvergleich mit Wildcard-Zeichen 145
Zeigerstruktur in einer Datei 229
Zeitdifferenz berechnen 83

Zeit prüfen 79
zentrieren von Texten 84
Zeitverhalten 13
Zwischencode 17